**Assignment : 1**

**Problem Statement:** Write a non-recursive and recursive program to calculate Fibonacci numbers and analyse theirtime and space complexity.

**# 1. Non-Recursive Fibonacci with User Input:**

def fibonacci\_non\_recursive(n):

# Handle edge cases

if n < 0:

return "Input must be a non-negative integer."

elif n == 0:

return 0

elif n == 1:

return 1

# Initialize base Fibonacci numbers

a, b = 0, 1

for i in range(2, n + 1):

fib = a + b

a = b

b = fib

return b

# Get input from the user and ensure valid input

try:

n = int(input("Enter a non-negative integer for non-recursive Fibonacci: "))

print(f"Non-recursive Fibonacci of {n}: {fibonacci\_non\_recursive(n)}")

except ValueError:

print("Invalid input! Please enter a valid non-negative integer.")

**OUTPUT:**

Enter a non-negative integer for non-recursive Fibonacci: 10

Non-recursive Fibonacci of 10: 55

**#2. Recursive Fibonacci with User Input:**

def fibonacci\_recursive(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

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

# Get input from the user

n = int(input("Enter a number for recursive Fibonacci: "))

print(f"Recursive Fibonacci of {n}: {fibonacci\_recursive(n)}")

**OUTPUT:**

Enter a number for recursive Fibonacci: 2

Recursive Fibonacci of 2: 1

**Assignment : 2**

**Problem Statement:** Write a program to implement Huffman Encoding using a greedy strategy.

import heapq

# Node structure for Huffman Tree

class HuffmanNode:

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

self.char = char

self.freq = freq

self.left = None

self.right = None

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

return self.freq < other.freq

def generate\_codes(root, current\_code, codes):

if root is None:

return

if root.char is not None:

codes[root.char] = current\_code

generate\_codes(root.left, current\_code + "0", codes)

generate\_codes(root.right, current\_code + "1", codes)

def build\_huffman\_tree(frequency):

heap = []

for char, freq in frequency.items():

heapq.heappush(heap, HuffmanNode(char, freq))

while len(heap) > 1:

node1 = heapq.heappop(heap)

node2 = heapq.heappop(heap)

merged = HuffmanNode(None, node1.freq + node2.freq)

merged.left = node1

merged.right = node2

heapq.heappush(heap, merged)

return heapq.heappop(heap)

def calculate\_frequency(data):

frequency = {}

for char in data:

if char not in frequency:

frequency[char] = 0

frequency[char] += 1

return frequency

def huffman\_encoding(data):

if not data:

return "Input data is empty.", None

frequency = calculate\_frequency(data)

huffman\_tree\_root = build\_huffman\_tree(frequency)

codes = {}

generate\_codes(huffman\_tree\_root, "", codes)

encoded\_data = "".join([codes[char] for char in data])

return encoded\_data, huffman\_tree\_root

def huffman\_decoding(encoded\_data, huffman\_tree\_root):

if not encoded\_data or huffman\_tree\_root is None:

return "Cannot decode. Either the data is empty or the tree is invalid."

decoded\_data = ""

current\_node = huffman\_tree\_root

for bit in encoded\_data:

if bit == '0':

current\_node = current\_node.left

else:

current\_node = current\_node.right

if current\_node.left is None and current\_node.right is None:

decoded\_data += current\_node.char

current\_node = huffman\_tree\_root

return decoded\_data

# Driver code for user input

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

data = input("Enter data to encode using Huffman coding: ")

encoded\_data, huffman\_tree\_root = huffman\_encoding(data)

if huffman\_tree\_root is not None:

print(f"Encoded Data: {encoded\_data}")

decoded\_data = huffman\_decoding(encoded\_data, huffman\_tree\_root)

print(f"Decoded Data: {decoded\_data}")

else:

print(encoded\_data) # Error message if input is invalid

**OUTPUT:**

Enter data to encode using Huffman coding: Hello World!

Encoded Data: 1110110001011011101111110100001100001

Decoded Data: Hello World!

**Assignment : 3**

**Problem Statement:** Write a program to solve the fractional knapsack problem using a greedy method.

# Class to represent an item with value and weight

class Item:

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

self.value = value

self.weight = weight

# Function to calculate the maximum value that can be carried

def fractional\_knapsack(items, capacity):

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

items.sort(key=lambda item: item.value / item.weight, reverse=True)

total\_value = 0.0 # To store the total value

for item in items:

if capacity >= item.weight:

# If the item can fit in the remaining capacity, take it all

capacity -= item.weight

total\_value += item.value

else:

# Otherwise, take the fraction of the item that fits

fraction = capacity / item.weight

total\_value += item.value \* fraction

break # The knapsack is full

return total\_value

# Driver code

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

# User input for number of items

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

# Initialize the items list

items = []

# Get user input for each item

for i in range(n):

value = float(input(f"Enter value of item {i + 1}: "))

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

items.append(Item(value, weight))

# User input for the capacity of the knapsack

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

# Calculate and print the maximum value

max\_value = fractional\_knapsack(items, capacity)

print(f"Maximum value we can obtain = {max\_value}")

**OUTPUT:**

Enter the number of items: 3

Enter value of item 1: 10

Enter weight of item 1: 100

Enter value of item 2: 60

Enter weight of item 2: 20

Enter value of item 3: 120

Enter weight of item 3: 30

Enter the capacity of the knapsack: 50

Maximum value we can obtain = 180.0

**Assignment : 4**

**Problem Statement:** Write a program to solve the 0-1 knapsack problem using dynamic programming or a branch and bound strategy.

**# This is first approach i.e dynamic programming...**

# Function to solve 0-1 Knapsack problem using Dynamic Programming

def knapsack\_dp(weights, values, capacity):

n = len(values)

# Create a DP table with size (n+1) x (capacity+1)

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

# Fill the DP table

for i in range(1, n + 1):

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

if weights[i - 1] <= w:

# Either include the item or exclude it

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

else:

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

# Return the maximum value for the given capacity

return dp[n][capacity]

# Driver code

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

# User input for the number of items

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

# Initialize lists for values and weights

values = []

weights = []

# Get user input for each item

for i in range(n):

value = float(input(f"Enter value of item {i + 1}: "))

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

values.append(value)

weights.append(weight)

# User input for the capacity of the knapsack

capacity = int(input("Enter the capacity of the knapsack (as an integer): "))

# Calculate and print the maximum value

max\_value = knapsack\_dp(weights, values, capacity)

print(f"Maximum value in knapsack = {max\_value}")

**OUTPUT:**

Enter the number of items: 3

Enter value of item 1: 60

Enter weight of item 1: 10

Enter value of item 2: 100

Enter weight of item 2: 20

Enter value of item 3: 120

Enter weight of item 3: 30

Enter the capacity of the knapsack (as an integer): 50

Maximum value in knapsack = 220.0

**# This is second approach i.e Branch and Bound**

from queue import PriorityQueue

# Node structure for Branch and Bound

class Node:

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

self.level = level

self.profit = profit

self.weight = weight

self.bound = bound

# For priority queue (max heap) comparison

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

return self.bound > other.bound

# Function to calculate upper bound

def calculate\_bound(node, n, capacity, values, weights):

if node.weight >= capacity:

return 0

profit\_bound = node.profit

j = node.level + 1

total\_weight = node.weight

while j < n and total\_weight + weights[j] <= capacity:

total\_weight += weights[j]

profit\_bound += values[j]

j += 1

if j < n:

profit\_bound += (capacity - total\_weight) \* (values[j] / weights[j])

return profit\_bound

# Function to solve 0-1 Knapsack problem using Branch and Bound

def knapsack\_bb(values, weights, capacity):

n = len(values)

q = PriorityQueue()

# Sort items by value-to-weight ratio

items = sorted(range(n), key=lambda i: values[i] / weights[i], reverse=True)

sorted\_weights = [weights[i] for i in items]

sorted\_values = [values[i] for i in items]

# Create a dummy node and insert it into the queue

u = Node(-1, 0, 0, 0)

v = Node(0, 0, 0, 0)

u.bound = calculate\_bound(u, n, capacity, sorted\_values, sorted\_weights)

q.put(u)

max\_profit = 0

while not q.empty():

u = q.get() # Get the node with the highest bound

if u.bound > max\_profit:

# Branching: Exclude or include the next item

v.level = u.level + 1

v.weight = u.weight + sorted\_weights[v.level]

v.profit = u.profit + sorted\_values[v.level]

# Check if including the item is feasible

if v.weight <= capacity and v.profit > max\_profit:

max\_profit = v.profit

# Calculate bound for including the item

v.bound = calculate\_bound(v, n, capacity, sorted\_values, sorted\_weights)

if v.bound > max\_profit:

q.put(v)

# Do the same for excluding the item

v.weight = u.weight

v.profit = u.profit

v.bound = calculate\_bound(v, n, capacity, sorted\_values, sorted\_weights)

if v.bound > max\_profit:

q.put(v)

return max\_profit

# Driver code

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

# User-defined input for values, weights, and capacity

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

values = []

weights = []

for i in range(n):

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

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

values.append(value)

weights.append(weight)

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

max\_value = knapsack\_bb(values, weights, capacity)

print(f"Maximum value in knapsack = {max\_value}")

**OUTPUT:**

Enter the number of items: 3

Enter value of item 1: 60

Enter weight of item 1: 10

Enter value of item 2: 100

Enter weight of item 2: 20

Enter value of item 3: 120

Enter weight of item 3: 30

Enter the capacity of the knapsack: 50

Maximum value in knapsack = 220

**Assignment : 5**

**Problem Statement:** Design an n x n matrix with the first queen already placed. Use backtracking to place the remaining queens and generate the final n-queens matrix.

**# Function to check if it's safe to place a queen at (row, col)**

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

# Check this column on upper side

for i in range(row):

if board[i][col] == "Q":

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] == "Q":

return False

# Check upper diagonal on right side

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

if board[i][j] == "Q":

return False

return True

# Backtracking function to solve the N-Queens problem

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

# Base case: If all queens are placed

if row >= len(board):

print\_board(board)

return True # Found one solution

# Try placing the queen in all columns of this row

for col in range(len(board)):

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

# Place the queen

board[row][col] = "Q"

# Recur to place the rest of the queens

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

return True # Found a valid solution

# If placing queen in this column doesn't lead to a solution, backtrack

board[row][col] = "." # Use '.' to represent empty spaces

return False # No valid solution for this row

# Function to print the chessboard

def print\_board(board):

for row in board:

print(" ".join(row))

print()

# Driver code

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

N = int(input("Enter the size of the chessboard (N): "))

board = [["." for \_ in range(N)] for \_ in range(N)] # N x N chessboard with '.' for empty spaces

# Start solving from the first row

if not solve\_n\_queens(board, 0):

print("No solution exists")

**OUTPUT:**

**1) Enter the size of the chessboard (N): 8**

**Q . . . . . . .**

**. . . . Q . . .**

**. . . . . . . Q**

**. . . . . Q . .**

**. . Q . . . . .**

**. . . . . . Q .**

**. Q . . . . . .**

**. . . Q . . . .**

**2) Enter the size of the chessboard (N): 4**

**. Q . .**

**. . . Q**

**Q . . .**

**. . Q .**

**Assignment : 6**

**Problem Statement:** Write a program for analysis of quick sort by using deterministic and randomized variant**.**

**# 1 Deterministic sorting...**

import time

# Function to partition the array

def partition(arr, low, high):

pivot = arr[high] # Last element as pivot

i = low - 1

for j in range(low, high):

if arr[j] <= pivot:

i += 1

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

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

# Deterministic QuickSort function

def quicksort\_deterministic(arr, low, high):

if low < high:

pi = partition(arr, low, high)

quicksort\_deterministic(arr, low, pi - 1)

quicksort\_deterministic(arr, pi + 1, high)

# Main execution

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

# User-defined input for array

user\_input = input("Enter the numbers to be sorted, separated by spaces: ")

arr = list(map(int, user\_input.split())) # Convert input string to list of integers

start = time.time()

quicksort\_deterministic(arr, 0, len(arr) - 1)

end = time.time()

print(f"Sorted array: {arr}")

print(f"Deterministic QuickSort Time: {end - start} seconds")

**OUTPUT:**

**Enter the numbers to be sorted, separated by spaces: 20 39 40 98**

**Sorted array: [20, 39, 40, 98]**

**Deterministic QuickSort Time: 0.0 seconds**

**# 2 Randomized Sort**

import time

import random

# Function to partition the array with random pivot

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

random\_pivot = random.randint(low, high)

arr[random\_pivot], arr[high] = arr[high], arr[random\_pivot] # Swap random pivot with last element

return partition(arr, low, high)

# Function to partition the array

def partition(arr, low, high):

pivot = arr[high] # Last element as pivot

i = low - 1

for j in range(low, high):

if arr[j] <= pivot:

i += 1

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

arr[i + 1], arr[high] = arr[high], arr[i + 1]

return i + 1

# Randomized QuickSort function

def quicksort\_randomized(arr, low, high):

if low < high:

pi = randomized\_partition(arr, low, high)

quicksort\_randomized(arr, low, pi - 1)

quicksort\_randomized(arr, pi + 1, high)

# Main execution

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

# User input for the array

user\_input = input("Enter numbers separated by spaces: ")

arr = list(map(int, user\_input.split())) # Convert input string to a list of integers

start = time.time()

quicksort\_randomized(arr, 0, len(arr) - 1)

end = time.time()

print(f"Sorted array: {arr}")

print(f"Randomized QuickSort Time: {end - start} seconds")

**OUTPUT:**

**Enter numbers separated by spaces: 20 18 2 3 6**

**Sorted array: [2, 3, 6, 18, 20]**

**Randomized QuickSort Time: 0.0 seconds**