**LP-III**

**1)write a program non-recursive and recursive program to calculate fibonacci numbers and analyse there time and space complexity**

**For non recursive in cpp**

**#include <iostream>**

**int fibonacci\_iterative(int n) {**

**if (n <= 0)**

**return 0;**

**else if (n == 1)**

**return 1;**

**int prev = 0;**

**int current = 1;**

**for (int i = 2; i <= n; ++i) {**

**int next\_fib = prev + current;**

**prev = current;**

**current = next\_fib;**

**}**

**return current;**

**}**

**int main() {**

**int n = 10; // Change this to the desired Fibonacci number you want to calculate.**

**int result = fibonacci\_iterative(n);**

**std::cout << "Fibonacci(" << n << ") = " << result << std::endl;**

**return 0;**

**}**

**For recursive**

**#include <iostream>**

**int fibonacci\_recursive(int n) {**

**if (n <= 0)**

**return 0;**

**else if (n == 1)**

**return 1;**

**else**

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

**}**

**int main() {**

**int n = 10; // Change this to the desired Fibonacci number you want to calculate.**

**int result = fibonacci\_recursive(n);**

**std::cout << "Fibonacci(" << n << ") = " << result << std::endl;**

**return 0;**

**}**

**//HA CODE RUN HOT NAHI**

**2)write a program to implement Huffman encoding using greedy strategy**

**import heapq**

**from collections import defaultdict, Counter**

**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 build\_huffman\_tree(data):**

**frequency = Counter(data)**

**min\_heap = [HuffmanNode(char, freq) for char, freq in frequency.items()]**

**heapq.heapify(min\_heap)**

**while len(min\_heap) > 1:**

**left = heapq.heappop(min\_heap)**

**right = heapq.heappop(min\_heap)**

**parent = HuffmanNode(None, left.freq + right.freq)**

**parent.left = left**

**parent.right = right**

**heapq.heappush(min\_heap, parent)**

**return min\_heap[0]**

**def build\_huffman\_codes(root, current\_code, huffman\_codes):**

**if root is None:**

**return**

**if root.char is not None:**

**huffman\_codes[root.char] = current\_code**

**build\_huffman\_codes(root.left, current\_code + '0', huffman\_codes)**

**build\_huffman\_codes(root.right, current\_code + '1', huffman\_codes)**

**def huffman\_encoding(data):**

**if not data:**

**return '', None**

**root = build\_huffman\_tree(data)**

**huffman\_codes = {}**

**build\_huffman\_codes(root, '', huffman\_codes)**

**encoded\_data = ''.join(huffman\_codes[char] for char in data)**

**return encoded\_data, root**

**def huffman\_decoding(encoded\_data, root):**

**if not encoded\_data or root is None:**

**return ''**

**current = root**

**decoded\_data = []**

**for bit in encoded\_data:**

**if bit == '0':**

**current = current.left**

**else:**

**current = current.right**

**if current.char is not None:**

**decoded\_data.append(current.char)**

**current = root**

**return ''.join(decoded\_data)**

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

**# Example usage**

**data = "hello, world!"**

**encoded\_data, tree = huffman\_encoding(data)**

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

**decoded\_data = huffman\_decoding(encoded\_data, tree)**

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

**3)write a program to solve a fractional knapsack problem using a greedy method**

**#include <iostream>**

**#include <vector>**

**#include <algorithm>**

**struct Item {**

**int weight;**

**int value;**

**double value\_per\_weight;**

**Item(int w, int v) : weight(w), value(v) {**

**value\_per\_weight = static\_cast<double>(v) / w;**

**}**

**};**

**bool compareItems(const Item& a, const Item& b) {**

**return a.value\_per\_weight > b.value\_per\_weight;**

**}**

**double fractionalKnapsack(std::vector<Item>& items, int capacity) {**

**double max\_value = 0.0;**

**// Sort items by value per weight in descending order**

**std::sort(items.begin(), items.end(), compareItems);**

**for (const Item& item : items) {**

**if (capacity == 0) {**

**break; // Knapsack is full**

**}**

**int weight\_to\_take = std::min(item.weight, capacity);**

**max\_value += weight\_to\_take \* item.value\_per\_weight;**

**capacity -= weight\_to\_take;**

**}**

**return max\_value;**

**}**

**int main() {**

**std::vector<Item> items = {**

**{10, 60},**

**{20, 100},**

**{30, 120}**

**};**

**int capacity = 50;**

**double max\_value = fractionalKnapsack(items, capacity);**

**std::cout << "Maximum value obtained: " << max\_value << std::endl;**

**return 0;**

**}**

**4)Write a program to solve 0-1 knapsack problems using dynamic programming or branch and bond strategy.**

**#include <iostream>**

**#include <vector>**

**int knapsackDP(int capacity, std::vector<int>& weights, std::vector<int>& values, int n) {**

**std::vector<std::vector<int>> dp(n + 1, std::vector<int>(capacity + 1, 0));**

**for (int i = 0; i <= n; i++) {**

**for (int w = 0; w <= capacity; w++) {**

**if (i == 0 || w == 0) {**

**dp[i][w] = 0;**

**} else if (weights[i - 1] <= w) {**

**dp[i][w] = std::max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);**

**} else {**

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

**}**

**}**

**}**

**return dp[n][capacity];**

**}**

**int main() {**

**std::vector<int> values = {60, 100, 120};**

**std::vector<int> weights = {10, 20, 30};**

**int capacity = 50;**

**int n = values.size();**

**int max\_value = knapsackDP(capacity, weights, values, n);**

**std::cout << "Maximum value: " << max\_value << std::endl;**

**return 0;**

**}**

**5)Design n-queens matrix having first Queen placed.Using backtracking to place remaining queens to generate final n-queens matrix**

**#include <iostream>**

**#include <vector>**

**bool isSafe(const std::vector<std::vector<int>>& board, int row, int col, int n) {**

**// Check if there is a queen in the same column**

**for (int i = 0; i < row; i++) {**

**if (board[i][col] == 1)**

**return false;**

**}**

**// Check upper left diagonal**

**for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {**

**if (board[i][j] == 1)**

**return false;**

**}**

**// Check upper right diagonal**

**for (int i = row, j = col; i >= 0 && j < n; i--, j++) {**

**if (board[i][j] == 1)**

**return false;**

**}**

**return true;**

**}**

**bool solveNQueens(std::vector<std::vector<int>>& board, int row, int n) {**

**if (row == n) {**

**// All queens are placed, and the solution is found.**

**return true;**

**}**

**for (int col = 0; col < n; col++) {**

**if (isSafe(board, row, col, n)) {**

**board[row][col] = 1; // Place the queen**

**// Recur to place the rest of the queens**

**if (solveNQueens(board, row + 1, n))**

**return true;**

**board[row][col] = 0; // If no safe spot found, backtrack**

**}**

**}**

**return false; // No solution found**

**}**

**void printBoard(const std::vector<std::vector<int>>& board, int n) {**

**for (int i = 0; i < n; i++) {**

**for (int j = 0; j < n; j++) {**

**std::cout << board[i][j] << " ";**

**}**

**std::cout << std::endl;**

**}**

**}**

**int main() {**

**int n;**

**std::cout << "Enter the value of n: ";**

**std::cin >> n;**

**std::vector<std::vector<int>> board(n, std::vector<int>(n, 0));**

**// Place the first queen in the first row (row 0)**

**board[0][0] = 1;**

**if (solveNQueens(board, 1, n)) {**

**std::cout << "N-Queens solution found:" << std::endl;**

**printBoard(board, n);**

**} else {**

**std::cout << "No solution found for N-Queens." << std::endl;**

**}**

**return 0;**

**}**