**Assignment -1**

public class FibonacciIterative {

public static long calculateFibonacci(int n) {

if (n <= 1) {

return n;

}

long fibNMinus2 = 0;

long fibNMinus1 = 1;

long fibN = 0;

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

fibN = fibNMinus1 + fibNMinus2;

fibNMinus2 = fibNMinus1;

fibNMinus1 = fibN;

}

return fibN;

}

public static void main(String[] args) {

int n = 10; // Calculate the 10th Fibonacci number

long startTime = System.nanoTime();

long result = calculateFibonacci(n);

long endTime = System.nanoTime();

System.out.println("Fibonacci(" + n + ") = " + result);

System.out.println("Time taken: " + (endTime - startTime) + " nanoseconds");

}

}

//Output :



public class FibonacciRecursive {

public static long calculateFibonacci(int n) {

if (n <= 1) {

return n;

}

return calculateFibonacci(n - 1) + calculateFibonacci(n - 2);

}

public static void main(String[] args) {

int n = 10; // Calculate the 10th Fibonacci number

long startTime = System.nanoTime();

long result = calculateFibonacci(n);

long endTime = System.nanoTime();

System.out.println("Fibonacci(" + n + ") = " + result);

System.out.println("Time taken: " + (endTime - startTime) + " nanoseconds");

}

}

//Output:



**Assignment -2**

import java.util.Comparator;

import java.util.PriorityQueue;

class HuffmanNode {

char data;

int frequency;

HuffmanNode left, right;

HuffmanNode(char data, int frequency) {

this.data = data;

this.frequency = frequency;

left = right = null;

}

}

class MyComparator implements Comparator<HuffmanNode> {

public int compare(HuffmanNode x, HuffmanNode y) {

return x.frequency - y.frequency;

}

}

class HuffmanEncoding {

public static void printCodes(HuffmanNode root, String code) {

if (root == null) {

return;

}

if (root.data != '$') {

System.out.println(root.data + ":" + code);

}

printCodes(root.left, code + "0");

printCodes(root.right, code + "1");

}

public static void buildHuffmanTree(char[] data, int[] freq, int n) {

PriorityQueue<HuffmanNode> minHeap = new PriorityQueue<>(n, new MyComparator());

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

HuffmanNode node = new HuffmanNode(data[i], freq[i]);

minHeap.add(node);

}

while (minHeap.size() > 1) {

HuffmanNode left = minHeap.poll();

HuffmanNode right = minHeap.poll();

HuffmanNode parent = new HuffmanNode('$', left.frequency + right.frequency);

parent.left = left;

parent.right = right;

minHeap.add(parent);

}

HuffmanNode root = minHeap.poll();

printCodes(root, "");

}

public static void main(String[] args) {

char[] data = { 'a', 'b', 'c', 'd', 'e', 'f' };

int[] freq = { 5, 9, 12, 13, 16, 45 };

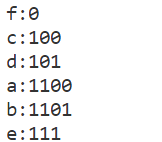
int n = data.length;

buildHuffmanTree(data, freq, n);

}

}

//Output:



**Assignment -3**

import java.util.Arrays;

import java.util.Comparator;

class Item {

int weight;

int value;

double valuePerWeight;

Item(int weight, int value) {

this.weight = weight;

this.value = value;

this.valuePerWeight = (double) value / weight;

}

}

class FractionalKnapsack {

public static double fractionalKnapsack(int capacity, Item[] items) {

Arrays.sort(items, Comparator.comparingDouble((Item item) -> item.valuePerWeight).reversed());

double totalValue = 0.0;

int remainingCapacity = capacity;

for (Item item : items) {

if (item.weight <= remainingCapacity) {

totalValue += item.value;

remainingCapacity -= item.weight;

} else {

totalValue += (item.valuePerWeight \* remainingCapacity);

break;

}

}

return totalValue;

}

public static void main(String[] args) {

int capacity = 50;

Item[] items = {

new Item(10, 60),

new Item(20, 100),

new Item(30, 120)

};

double maxValue = fractionalKnapsack(capacity, items);

System.out.println("Maximum value that can be obtained: " + maxValue);

}

}

//Output :



**Assignment -4**

**1)**

Import java.util.PriorityQueue;

class Node implements Comparable<Node> {

int level;

int profit;

int weight;

double bound;

Node(int level, int profit, int weight) {

this.level = level;

this.profit = profit;

this.weight = weight;

}

@Override

public int compareTo(Node other) {

return Double.compare(other.bound, this.bound);

}

}

class BranchAndBoundKnapsack {

public static double knapsack(int capacity, int[] weights, int[] values, int n) {

PriorityQueue<Node> priorityQueue = new PriorityQueue<>();

Node u, v;

// Initialize the root node.

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

u.bound = computeBound(u, capacity, weights, values, n);

double maxProfit = 0.0;

// Add the root node to the priority queue.

priorityQueue.add(u);

while (!priorityQueue.isEmpty()) {

// Get the highest bound node.

u = priorityQueue.poll();

if (u.bound > maxProfit) {

int level = u.level + 1;

// Include the next item.

v = new Node(level, u.profit + values[level], u.weight + weights[level]);

v.bound = computeBound(v, capacity, weights, values, n);

if (v.weight <= capacity && v.profit > maxProfit) {

maxProfit = v.profit;

}

if (v.bound > maxProfit) {

priorityQueue.add(v);

}

// Exclude the next item.

v = new Node(level, u.profit, u.weight);

v.bound = computeBound(v, capacity, weights, values, n);

if (v.bound > maxProfit) {

priorityQueue.add(v);

}

}

}

return maxProfit;

}

public static double computeBound(Node node, int capacity, int[] weights, int[] values, int n) {

if (node.weight >= capacity) {

return 0;

}

double bound = node.profit;

int j = node.level + 1;

int totalWeight = node.weight;

while (j < n && totalWeight + weights[j] <= capacity) {

totalWeight += weights[j];

bound += values[j];

j++;

}

if (j < n) {

bound += (capacity - totalWeight) \* ((double) values[j] / weights[j]);

}

return bound;

}

public static void main(String[] args) {

int capacity = 10;

int[] weights = {2, 1, 3, 2};

int[] values = {12, 10, 20, 15};

int n = weights.length;

double maxValue = knapsack(capacity, weights, values, n);

System.out.println("Maximum value that can be obtained: " + maxValue);

}

}

//Output :



2)

public class ZeroOneKnapsack {

public static int knapsack(int capacity, int[] weights, int[] values, int n) {

int[][] dp = new int[n + 1][capacity + 1];

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] = Math.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];

}

public static void main(String[] args) {

int capacity = 10;

int[] weights = {2, 1, 3, 2};

int[] values = {12, 10, 20, 15};

int n = weights.length;

int maxValue = knapsack(capacity, weights, values, n);

System.out.println("Maximum value that can be obtained: " + maxValue);

}

}

//Output :



**Assignment -5**

public class NQueensWithFirstQueenPlaced {

public static void printBoard(int[][] board) {

int n = board.length;

System.out.println();

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

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

System.out.print(board[i][j] + " ");

}

System.out.println();

}

}

public static boolean isSafe(int[][] board, int row, int col) {

int n = board.length;

// Check left side of the current row

for (int i = 0; i < col; i++) {

if (board[row][i] == 1) {

return false;

}

}

// Check upper diagonal on the left side

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

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

return false;

}

}

// Check lower diagonal on the left side

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

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

return false;

}

}

return true;

}

public static boolean solveNQueens(int[][] board, int col) {

int n = board.length;

if (col >= n) {

// All queens are placed, return true

return true;

}

// Try placing the queen in each row of the current column

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

if (isSafe(board, i, col)) {

// Place the queen

board[i][col] = 1;

// Recur to place the rest of the queens

if (solveNQueens(board, col + 1)) {

return true;

}

// If placing the queen in board[i][col] doesn't lead to a solution, backtrack

board[i][col] = 0;

}

}

return false;

}

public static void main(String[] args) {

int n = 8; // Change 'n' to the desired board size

int[][] board = new int[n][n];

// Place the first queen at (0, 0)

board[0][0] = 1;

// Call the backtracking function to solve the rest of the board

if (solveNQueens(board, 1)) {

System.out.println("Solution exists:");

printBoard(board);

} else {

System.out.println("No solution exists.");

}

}

}

//Output :

