DAA Practical 1

```
import java.util.Scanner;

public class A1_1 {

    // --- Recursive function to print Fibonacci series ---
    public static void printFibonacciRecursive(int a, int b, int n) {
        if (n == 0)
            return;
        System.out.print(a + " ");
        printFibonacciRecursive(b, a + b, n - 1);
    }

    // --- Iterative function to print Fibonacci series ---
    public static void printFibonacciIterative(int n) {
        int a = 0, b = 1;
        for (int i = 1; i <= n; i++) {
            System.out.print(a + " ");
            int c = a + b;
            a = b;
            b = c;
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of terms: ");
        int n = sc.nextInt();
        sc.close();
```

```java
        if (n <= 0) {
            System.out.println("Please enter a positive number.");
            return;
        }

        // --- Iterative Approach ---
        System.out.println("\nIterative Fibonacci Series:");
        printFibonacciIterative(n);
        System.out.println("\nTime Complexity: O(n)");
        System.out.println("Space Complexity: O(1)");

        // --- Recursive Approach ---
        System.out.println("\n\nRecursive Fibonacci Series:");
        printFibonacciRecursive(0, 1, n);
        System.out.println("\nTime Complexity: O(n)");
        System.out.println("Space Complexity: O(n) (due to recursion stack)");
    }
}
```

OUYPUT:

```
Enter number of terms: 7

Iterative Fibonacci Series:
0 1 1 2 3 5 8
Time Complexity: O(n)
Space Complexity: O(1)



Recursive Fibonacci Series:
0 1 1 2 3 5 8
Time Complexity: O(n)
Space Complexity: O(n) (due to recursion stack)
```

DAA Practical 1

```
package Daapractical;

import java.util.Scanner;

public class A1 {

   // --- Recursive Implementation ---
   /**
    * Calculates the nth Fibonacci number recursively.
    * Time Complexity: O(2^n)
    * Space Complexity: O(n) (due to recursion stack depth)
    */
   public static long fibonacciRecursive(int n) {
      if (n <= 1) {
         return n; // Base case: F(0)=0, F(1)=1
      }
      // Recurrence relation: F(n) = F(n-1) + F(n-2)
      return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
   }

   // --- Non-Recursive (Iterative) Implementation ---
   /**
    * Calculates the nth Fibonacci number non-recursively (iteratively).
    * Time Complexity: O(n)
    * Space Complexity: O(1)
    */
   public static long fibonacciNonRecursive(int n) {
      if (n <= 1) {
         return n; // Base case
      }
```

```java
        long fnm1 = 0; // fnm2:=0
        long fnm2 = 1; // fnm1:=1
        long fn = 0;

        // for i:=2 to n do
        for (int i = 2; i <= n; i++) {
            fn = fnm1 + fnm2; // fn:=fnm1+fnm2
            fnm1 = fnm2; // fnm1:=fnm2
            fnm2 = fn; // fnm2:=fn
        }

        return fn;
    }

    // --- Main method for demonstration ---
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number n to find the nth Fibonacci number: ");
        int n = scanner.nextInt();
        scanner.close();

        if (n < 0) {
            System.out.println("Fibonacci number is not defined for negative input.");
            return;
        }

        // 1. Non-Recursive Solution
        long resultNonRec = fibonacciNonRecursive(n);
        System.out.println("\nNon-Recursive Fibonacci of " + n + " is: " + resultNonRec);
        System.out.println("Time Complexity (Non-Recursive): O(n)");
```

```
        System.out.println("Space Complexity (Non-Recursive): O(1)");

        // 2. Recursive Solution
        long resultRec = fibonacciRecursive(n);
        System.out.println("\nRecursive Fibonacci of " + n + " is: " + resultRec);
        // The time complexity is exponential, proportional to 2^n
        System.out.println("Time Complexity (Recursive): O(2^n)");
        // Space complexity is O(n) due to the depth of the recursion stack.
        System.out.println("Space Complexity (Recursive): O(n)");
    }
}
```

OUTPUT:

```
Enter the number n to find the nth Fibonacci number: 7

Non-Recursive Fibonacci of 7 is: 13
Time Complexity (Non-Recursive): O(n)
Space Complexity (Non-Recursive): O(1)

Recursive Fibonacci of 7 is: 13
Time Complexity (Recursive): O(2^n)
Space Complexity (Recursive): O(n)
```

DAA Practical 2

```java
package Daapractical;

import java.util.PriorityQueue;
import java.util.HashMap;
import java.util.Map;
import java.util.Comparator;

public class A2 {

    // A node in the Huffman Tree
    static class HuffmanNode {
        char data;
        int frequency;
        HuffmanNode left;
        HuffmanNode right;

        public HuffmanNode(char data, int frequency) {
            this.data = data;
            this.frequency = frequency;
            this.left = null;
            this.right = null;
        }

        public HuffmanNode(int frequency, HuffmanNode left, HuffmanNode right) {
            this.data = '\0'; // Internal node has null character
            this.frequency = frequency;
            this.left = left;
            this.right = right;
        }
    }
```

```java
// Comparator for the PriorityQueue (Min Heap)
static class FrequencyComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y) {
        // Compare nodes based on their frequency (Min-Heap based on frequency)
        return x.frequency - y.frequency;
    }
}

/**
 * Builds the Huffman Tree using a greedy approach (Min-Heap).
 */
public static HuffmanNode buildHuffmanTree(HashMap<Character, Integer> charFrequencies) {
    // Create a Min Heap (PriorityQueue)
    PriorityQueue<HuffmanNode> minHeap = new PriorityQueue<>(new FrequencyComparator());

    // i) Create a leaf node for each unique character and build a min heap of all
    // leaf nodes
    for (Map.Entry<Character, Integer> entry : charFrequencies.entrySet()) {
        minHeap.add(new HuffmanNode(entry.getKey(), entry.getValue()));
    }

    // Repeat steps #2 and #3 until the heap contains only one node
    while (minHeap.size() > 1) {
        // ii) Extract two nodes with the minimum frequency from the min heap
        HuffmanNode x = minHeap.poll();
        HuffmanNode y = minHeap.poll();

        // iii) Create a new internal node with a frequency equal to the sum of the two
        // nodes frequencies
        // Make the first extracted node as its left child and the other extracted node
```

```java
            // as its right child
            HuffmanNode newNode = new HuffmanNode(x.frequency + y.frequency, x, y);

            // iv) Add this node to the min heap
            minHeap.add(newNode);
        }

        // The remaining node is the root node and the tree is complete
        return minHeap.poll();
    }

    /**
     * Traverses the Huffman Tree to generate codes for each character.
     * While moving to the left child, write '0' to the code.
     * While moving to the right child, write '1' to the code.
     * Print the code when a leaf node is encountered.
     */
    public static void generateCodes(HuffmanNode root, String code, Map<Character, String>
huffmanCodes) {
        if (root == null) {
            return;
        }

        if (root.left == null && root.right == null) {
            // Leaf node: a character node
            huffmanCodes.put(root.data, code);
            return;
        }

        // Go left (0)
        generateCodes(root.left, code + "0", huffmanCodes);
```

```
        // Go right (1)
        generateCodes(root.right, code + "1", huffmanCodes);
    }

    // --- Main method for demonstration ---
    public static void main(String[] args) {
        // Example frequencies from the lab manual
        HashMap<Character, Integer> frequencies = new HashMap<>();
        frequencies.put('a', 5);
        frequencies.put('b', 9);
        frequencies.put('c', 12);
        frequencies.put('d', 13);
        frequencies.put('e', 16);
        frequencies.put('f', 45);

        HuffmanNode root = buildHuffmanTree(frequencies);

        Map<Character, String> huffmanCodes = new HashMap<>();
        generateCodes(root, "", huffmanCodes);

        System.out.println("Huffman Codes:");
        System.out.println("Character | Code-Word");
        System.out.println("---------------------");
        for (Map.Entry<Character, String> entry : huffmanCodes.entrySet()) {
            System.out.printf("%-9s | %s\n", entry.getKey(), entry.getValue());
        }

        System.out.println("\nTime Complexity: O(n log n) where n is the number of unique
characters.");
        System.out.println("Space Complexity: O(n) to store the tree and codes.");
    }
```

}

OUTPUT:

Huffman Codes:

Character | Code-Word

----

a | 1100

b | 1101

c | 100

d | 101

e | 111

f | 0

Time Complexity: O(n log n) where n is the number of unique characters.

Space Complexity: O(n) to store the tree and codes.

DAA Practical 3

```java
package Daapractical;

import java.util.Arrays;
import java.util.Comparator;

public class A3 {

   // Class to represent an item in the Knapsack problem
   static class Item {
      int value;
      int weight;
      double ratio; // value/weight ratio

      public Item(int value, int weight) {
         this.value = value;
         this.weight = weight;
         this.ratio = (double) value / weight; // Calculate value-to-weight ratio
      }
   }
   /**
    * Solves the Fractional Knapsack problem using the greedy approach.
    */
   public static double getMaxValue(Item[] items, int capacity) {
      // Sort items by their value-to-weight ratio in descending order
      // O(n log n) time complexity
      Arrays.sort(items, new Comparator<Item>() {
         @Override
         public int compare(Item o1, Item o2) {
            // For descending order, compare o2.ratio to o1.ratio
            if (o2.ratio > o1.ratio)
```

```
            return 1;
        if (o2.ratio < o1.ratio)
            return -1;
        return 0;
    }
});

int currentWeight = 0;
double finalValue = 0.0;
// Iterate through the sorted items and add them to the knapsack
// O(n) time complexity
for (Item item : items) {
    if (currentWeight + item.weight <= capacity) {
        // Take the whole item
        currentWeight += item.weight;
        finalValue += item.value;
    } else {
        // Take a fraction of the item
        int remainingCapacity = capacity - currentWeight;

        // Fraction = (remainingCapacity / item.weight)
        double fraction = (double) remainingCapacity / item.weight;

        // Add the fraction's value
        finalValue += item.value * fraction;

        // Knapsack is full, so we stop
        break;
    }
}
```

```
        return finalValue;
    }

    // --- Main method for demonstration ---
    public static void main(String[] args) {
        // Example data
        Item[] items = {
            new Item(60, 10), // Value=60, Weight=10, Ratio=6.0
            new Item(100, 20), // Value=100, Weight=20, Ratio=5.0
            new Item(100, 50), // Value=100, Weight=30, Ratio=2.0
            new Item(200, 50) // Value=200, Weight=50, Ratio=4.0
        };
        int knapsackCapacity = 90; // W

        double maxValue = getMaxValue(items, knapsackCapacity);

        System.out.println("Maximum value in Knapsack for capacity " + knapsackCapacity + " is: "
            + String.format("%.2f", maxValue));

        // Total time is dominated by the sorting step
        System.out.println("\nTime Complexity: O(n log n) due to sorting");
        System.out.println("Space Complexity: O(1) (if sorting is in-place) or O(n) for storing
items/ratios.");
    }
}
```

OUTPUT:

Maximum value in Knapsack for capacity 90 is: 380.00

Time Complexity: O(n log n) due to sorting

Space Complexity: O(1) (if sorting is in-place) or O(n) for storing items/ratios.

DAA Practical 4

```
package Daapractical;

import java.util.*;

class A4{
  // Helper function to solve the knapsack problem recursively
  static int knapsackUtil(int[] wt, int[] val, int ind, int W, int[][] dp) {
    // Base case: If there are no items or the knapsack capacity is zero
    if (ind == 0) {
      if (wt[0] <= W) {
        // Include the item if its weight is within the capacity
        return val[0];
      } else {
        // Otherwise, exclude the item
        return 0;
      }
    }

    // If the result for this subproblem is already calculated, return it
    if (dp[ind][W] != -1) {
      return dp[ind][W];
    }

    // Calculate the maximum value when the current item is not taken
    int notTaken = 0 + knapsackUtil(wt, val, ind - 1, W, dp);

    // Calculate the maximum value when the current item is taken
    int taken = Integer.MIN_VALUE;
    if (wt[ind] <= W) {
      taken = val[ind] + knapsackUtil(wt, val, ind - 1, W - wt[ind], dp);
```

```
    }

    // Store and return the result for the current state
    dp[ind][W] = Math.max(notTaken, taken);
    return dp[ind][W];
  }

  // Function to solve the 0/1 Knapsack problem using dynamic programming
  static int knapsack(int[] wt, int[] val, int n, int W) {
    // Create a 2D DP array to store the maximum value for each subproblem
    int dp[][] = new int[n][W + 1];

    // Initialize the DP array with -1 to indicate that subproblems are not solved
    for (int row[] : dp) {
      Arrays.fill(row, -1);
    }

    // Call the recursive knapsackUtil function to solve the problem
    return knapsackUtil(wt, val, n - 1, W, dp);
  }

  public static void main(String args[]) {
    int wt[] = {3,2,5};
    int val[] = {30,40,60};
    int W = 6;
    int n = wt.length;

    // Calculate and print the maximum value of items the thief can steal
    System.out.println("The Maximum value of items the thief can steal is " + knapsack(wt, val, n,
W));
```

```java
        // Print time and space complexity
        System.out.println("\n--- Complexity Analysis ---");
        System.out.println("Time Complexity  : O(N * W)  Reason: There are N*W states therefore at max ‘N*W’ new problems will be solved.");
        System.out.println("Space Complexity : O(N * W) for memoization table Reason: We are using a recursion stack space(O(N)) and a 2D array ( O(N*W)).");
    }
}
```

OUTPUT:

The Maximum value of items the thief can steal is 70

--- Complexity Analysis ---

Time Complexity : O(N * W) Reason: There are N*W states therefore at max ‘N*W’ new problems will be solved.

Space Complexity : O(N * W) for memoization table Reason: We are using a recursion stack space(O(N)) and a 2D array (O(N*W)).

DAA Practical 5

```
package Daapractical;

import java.util.Arrays;
import java.util.Scanner;

public class NQueensBacktracking {

    private int N;
    private int[][] board;
    private int solutionCount = 0;

    public NQueensBacktracking(int n) {
        this.N = n;
        this.board = new int[n][n];
    }

    /**
     * Utility function to print the solution board (binary matrix).
     */
    private void printSolution() {
        solutionCount++;
        System.out.println("\nSolution " + solutionCount + ":");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(" " + board[i][j] + " ");
            }
            System.out.println();
        }
    }
```

```java
/**
 * Checks if a queen can be safely placed at board[row][col].
 */
private boolean isSafe(int row, int col) {
    int i, j;

    // 1. Check this row on the left side (for queens in previous columns) [cite:
    // 445]
    for (i = 0; i < col; i++) {
        if (board[row][i] == 1) {
            return false;
        }
    }

    // 2. Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    // 3. Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    return true;
}
```

```
/**
 * Recursive backtracking function to place queens column by column.
 */
private boolean solveNQUtil(int col) {
  // Base case: If all columns are processed (all queens placed), return true
  // [cite: 449, 450]
  if (col >= N) {
    printSolution();
    // Since N-Queens can have multiple solutions, return 'false' to find others, or
    // 'true' to stop after the first.
    // We use 'true' to indicate a solution was found in this branch, but the
    // calling function handles finding others.
    return true;
  }

  // If the current column already has a queen (the pre-placed one), skip all
  // other placements in this column.
  boolean isPrePlaced = false;
  for (int row = 0; row < N; row++) {
    if (board[row][col] == 1) {
      isPrePlaced = true;
      break;
    }
  }

  if (isPrePlaced) {
    // Only continue the search from the next column
    return solveNQUtil(col + 1);
  }

  boolean foundSolution = false;
```

```
    // Try all rows in the current column [cite: 451]
    for (int i = 0; i < N; i++) {
        // If the queen can be placed safely in this row [cite: 453]
        if (isSafe(i, col)) {

            // Place the queen [cite: 453]
            board[i][col] = 1;

            // Recursively check if placing the queen here leads to a solution [cite: 453]
            if (solveNQUtil(col + 1)) {
                foundSolution = true;
                // We DO NOT break here, to continue searching for other solutions
            }

            // If placing queen doesn't lead to a solution, then backtrack [cite: 456]
            board[i][col] = 0; // Unmark (Backtrack)
        }
    }

    return foundSolution;
}

/**
 * Main solver function to set up the initial queen and start backtracking.
 */
public void solveNQueens(int initialRow, int initialCol) {

    // Initialize the board for N x N
    for (int[] row : board) {
        Arrays.fill(row, 0);
```

```
    }

    // Step 1: Place the first queen
    if (initialRow >= 0 && initialRow < N && initialCol >= 0 && initialCol < N) {
      board[initialRow][initialCol] = 1;
    } else {
      System.out.println("Invalid initial position for the first Queen.");
      return;
    }

    // Start backtracking from the *first column (col=0)*
    solveNQUtil(0);

    if (solutionCount == 0) {
      System.out.println("Solution does not exist for N=" + N + " with the first queen placed at (" +
initialRow
          + ", " + initialCol + ")");
    } else {
      System.out.println("\nTotal solutions found: " + solutionCount);
    }

    System.out.println("Time Complexity (Worst Case): O(N!) [cite: 457]");
    System.out.println("Space Complexity: O(N^2) for the board array + O(N) for recursion stack.");
  }

  // --- Main method for demonstration ---
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter the board size N (e.g., 4, 8): ");
    int n = scanner.nextInt();
```

```
        System.out.print("Enter the row for the pre-placed Queen (0 to N-1, e.g., 1 for 4x4): ");
        int initialRow = scanner.nextInt();

        System.out.print("Enter the column for the pre-placed Queen (0 to N-1, e.g., 0 for 4x4): ");
        int initialCol = scanner.nextInt();
        scanner.close();

        NQueensBacktracking queenSolver = new NQueensBacktracking(n);
        queenSolver.solveNQueens(initialRow, initialCol);
    }
}
```

OUTPUT:

```
Enter the board size N (e.g., 4, 8): 4
Enter the row for the pre-placed Queen (0 to N-1, e.g., 1 for 4x4): 2
Enter the column for the pre-placed Queen (0 to N-1, e.g., 0 for 4x4): 1

Solution 1:
 0  0  1  0
 1  0  0  0
 0  1  0  0
 0  0  0  1

Total solutions found: 1
Time Complexity (Worst Case): O(N!) [cite: 457]
Space Complexity: O(N^2) for the board array + O(N) for recursion stack.
```