W6 - Data Structure & Algorithms

Problem 1: Sorting Algorithms with Intermediate Results

Overview: Implement Selection sort and Bubble sort algorithms. Display the array's state after each iteration of the outer loop.

Selection Sort Algorithm:

1. **Initialize Variables**:
   - Start with the first element as the minimum.
2. **Iterate Through Array**:
   - For each iteration of the outer loop, find the minimum element from the unsorted part and swap it with the first element of the unsorted part.
3. **Display Intermediate Results**:
   - After each outer loop iteration, display the current state of the array.

Bubble Sort Algorithm:

1. **Initialize Variables**:
   - Start from the first element of the array.
2. **Iterate and Swap**:
   - In each iteration of the outer loop, bubble up the largest element to the end of the array by pairwise swapping.
3. **Display Intermediate Results**:
   - After each outer loop iteration, display the current state of the array.

Code

```
public class SortExample {
    public static void main(String[] args) {
        int[] array = {5, 1, 9, 6, 2};
        selectionSort(array);
        // bubbleSort(array);
    }

    private static void selectionSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first element
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;

            // Print the current state of array
            printArray(arr);
        }
    }

    private static void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }

            // Print the current state of array
            printArray(arr);
        }
    }

    private static void printArray(int[] arr) {
        for (int value : arr) {
            System.out.print(value + " ");
        }
        System.out.println();
    }
}
```

Problem 2: Knapsack Problem

Overview: Given an array `V` storing the values of `N` items, an array `W` storing the weights of `N` items, and a knapsack capacity `C`, calculate and return the subset of `N` items that has the highest sum value, and its total weight does not exceed `C`.

Approach:

- Use Dynamic Programming to solve this problem.
- Create a 2D array `dp` of size `(N+1) x (C+1)` where `dp[i][j]` will store the maximum value that can be achieved with the first `i` items and a total weight not exceeding `j`.

Algorithm Steps:

1. **Initialization**:
   - Initialize `dp[0][j]` and `dp[i][0]` to `0` for all `i` and `j`.
2. **Build the DP Table**:
   - For each item `i` and each capacity `j`, calculate `dp[i][j]`.
   - If `W[i-1]` (weight of current item) is less than or equal to `j` (current capacity), then the item can be included or excluded.
   - The maximum value is the max of including the item and not including it.
3. **Find the Result**:
   - The answer is `dp[N][C]`.
4. **Code**:

```
public class KnapsackProblem {
   public static void main(String[] args) {
       int[] V = {60, 100, 120}; // Values
       int[] W = {10, 20, 30};   // Weights
       int C = 50;               // Knapsack Capacity
       System.out.println("Maximum value subset: " + knapsack(V, W, C));
   }

   private static int knapsack(int[] V, int[] W, int C) {
       int N = V.length;
       int[][] dp = new int[N + 1][C + 1];

       for (int i = 0; i <= N; i++) {
           for (int j = 0; j <= C; j++) {
               if (i == 0 || j == 0)
                   dp[i][j] = 0;
               else if (W[i - 1] <= j)
                   dp[i][j] = Math.max(V[i - 1] + dp[i - 1][j - W[i - 1]], dp[i - 1][j]);
               else
                   dp[i][j] = dp[i - 1][j];
           }
```

```
        }
        return dp[N][C];
    }
}
```

Problem 3: 8-Queens with Pruning

Overview: The 8-Queens problem involves placing eight queens on an 8x8 chessboard so that no two queens threaten each other. The solution requires that no two queens share the same row, column, or diagonal.

Pruning Approach:

- Use backtracking to systematically place queens on the board.
- Prune (skip) positions where a queen would be in conflict with already placed queens.
- The pruning is done by checking for conflicts along the same row, column, and diagonals.

Algorithm Steps:

1. **Start with an Empty Board**:
    - Represent the board as a 2D array or a single array tracking queen positions.
2. **Place Queens One by One**:
    - For each row, try placing a queen in each column and check for conflicts.
3. **Pruning**:
    - If a conflict is detected (same row, column, or diagonal), skip the current column and try the next one.
4. **Backtrack if Necessary**:
    - If no valid position is found in a row, backtrack to the previous row and move the queen to the next valid position.
5. **Repeat Until All Queens Are Placed**:
    - Continue this process until all eight queens are placed without conflicts.
6. **Code**:

```
public class EightQueens {

    final int N = 8;

    // Driver method
    public static void main(String args[]) {
        EightQueens Queen = new EightQueens();
        Queen.solveNQ();
    }

    /* A utility function to print solution */
    void printSolution(int board[][]) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j] + " ");
            System.out.println();
        }
    }

    /* A utility function to check if a queen can be placed on board[row][col] */
    boolean isSafe(int board[][], int row, int col) {
        int i, j;

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

        // 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;
    }

    /* A recursive utility function to solve N Queen problem */
    boolean solveNQUtil(int board[][], int col) {
        // base case: If all queens are placed
        if (col >= N)
            return true;

        // Consider this column and try placing this queen in all rows one by one
        for (int i = 0; i < N; i++) {
            // Check if the queen can be placed on board[i][col]
            if (isSafe(board, i, col)) {
                // Place this queen in board[i][col]
                board[i][col] = 1;

                // recur to place rest of the queens
                if (solveNQUtil(board, col + 1))
                    return true;

                // If placing queen in board[i][col] doesn't lead to a solution, then remove queen from board[i]
                board[i][col] = 0; // BACKTRACK
            }
        }

        // If the queen can not be placed in any row in this column col then return false
        return false;
    }

    /* This function solves the N Queen problem using Backtracking */
    boolean solveNQ() {
        int board[][] = new int[N][N];

        if (!solveNQUtil(board, 0)) {
            System.out.print("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }
}
```