Practical 4

```
public class dpKnapsak {
   public static int KnapsackTab(int val[], int wt[], int W) {
      int n = val.length;
      int dp[][] = new int [n+1][W+1];
      for(int i=0; i<dp.length; i++) {
         dp[i][0]=0;
      }
      for(int j=0; j<dp[0].length; j++) {
         dp[0][j]=0;
      }
      for (int i=1; i<n+1; i++){
         for (int j=1; j<W+1; j++){
            int v = val[i-1];
            int w = wt[i-1];
            if(w <= j) {
               int incProfit = v + dp[i-1][j-w];
               int excProfit = dp[i-1][j];
               dp[i][j] = Math.max(incProfit, excProfit);
            }else{
               int excProfit = dp[i-1][j];
               dp[i][j] = excProfit;
            }
         }
      }
      return dp[n][W];

   }

   public static void main (String args[]){
      int val[] = {15, 14, 10, 45, 30};
```

```
    int wt[] = {2, 5, 1, 3, 4};


    int W = 7;
    int dp[][] = new int[val.length+1][W+1];


    for(int i=0; i<dp.length; i++) {
      for( int j=0; j<dp[0].length; j++){
        dp[i][j] = -1;
      }
    }
    System.out.println(KnapsackTab(val, wt, W));
  }
}
```

Output :-

Final value 75

Practical 5

```
public class practical5 {
  // Function to check if a queen can be placed at board[row][col]
  public static boolean isSafe(int[][] board, int row, int col, int n) {
    // Check this column on the left of the current row
    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;
  }
  // Function to solve N-Queens problem using backtracking
  public static boolean solveNQueens(int[][] board, int row, int n) {
    // Base case: If all queens are placed
    if (row >= n) {
      return true;
    }
```

```
    // Try placing the queen in every column of the current row
    for (int col = 0; col < n; col++) {
        if (isSafe(board, row, col, n)) {
            // Place the queen
            board[row][col] = 1;
            // Recursively place the rest of the queens
            if (solveNQueens(board, row + 1, n)) {
                return true;
            }
            // If placing the queen at board[row][col] doesn't lead to a solution
            // backtrack: remove the queen
            board[row][col] = 0;
        }
    }
    // If no queen can be placed in this row, return false
    return false;
}

// Function to print the solution board
public static void printBoard(int[][] board) {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[i].length; j++) {
            System.out.print(board[i][j] + " ");
        }
        System.out.println();
    }
}
// Function to initialize the board, place the first queen, and solve the problem
public static void nQueensWithFirstQueen(int n, int firstRow, int firstCol) {
    // Initialize an empty board
    int[][] board = new int[n][n];
```

```
        // Place the first queen at the given position
        board[firstRow][firstCol] = 1;

        // Start solving from the second row (since the first queen is already placed)
        if (!solveNQueens(board, firstRow + 1, n)) {
            System.out.println("No solution exists");
        } else {
            printBoard(board);
        }
    }

    // Main method
    public static void main(String[] args) {
        int n = 5; // Size of the chessboard (N-Queens)
        int firstRow = 0; // Row where the first queen is placed
        int firstCol = 0; // Column where the first queen is placed

        // Call the function to solve the N-Queens problem
        nQueensWithFirstQueen(n, firstRow, firstCol);
    }
}
```

Output:-

```
1 0 0 0 0
0 0 1 0 0
0 0 0 0 1
0 1 0 0 0
0 0 0 1 0
```