```java
Codeium: Refactor | Explain
public class Knaspsack01 {
    //Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value
    // in the knapsack.
    //In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and
    //weights associated with n items
    //respectively. Also given an integer W which represents knapsack capacity, find out the
    //maximum value subset of val[] such that
    //sum of the weights of this subset is smaller than or equal to W. You cannot break
    //an item, either pick the complete item, or
    //don't pick it (0-1 property).
    //Example: val[] = {60, 100, 120}; wt[] = {10, 20, 30}; W = 50 -> 220
//Approach: Recursion

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
//          int[] val = {60, 100, 120};
//          int[] wt = {10, 20, 30};
//          int W = 50;
        int[] val = {1, 2, 3};
        int[] wt = {4, 5, 1};
        int W = 4;
        System.out.println("knapsack01(val, wt, W) = " + knapsack01(val, wt, W));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int knapsack01(int[] val, int[] wt, int W) {
        int n = val.length;
        return knapsack01(val, wt, W, n);
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int knapsack01(int[] val, int[] wt, int W, int n) {
        //Base condition
        if (n == 0 || W == 0) {
            return 0;
        }

        //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
        if (wt[n - 1] > W) {
            return knapsack01(val, wt, W, n - 1);
        }

        //Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        return Math.max(val[n - 1] + knapsack01(val, wt, W - wt[n - 1], n - 1), knapsack01(val, wt, W, n - 1));
    }

```

```java
    //Approach: Memoization
    Codeium: Refactor | Explain | ×
    public static int knapsack01Memo(int[] val, int[] wt, int W) {
        int n = val.length;
        int[][] dp = new int[n + 1][W + 1];
        return knapsack01Memo(val, wt, W, n, dp);
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int knapsack01Memo(int[] val, int[] wt, int W, int n, int[][] dp) {
        //Base condition
        if (n == 0 || W == 0) {
            return 0;
        }

        if (dp[n][W] != 0) {
            return dp[n][W];
        }

        //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
        if (wt[n - 1] > W) {
            dp[n][W] = knapsack01Memo(val, wt, W, n - 1, dp);
            return dp[n][W];
        }

        //Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        dp[n][W] = Math.max(val[n - 1] + knapsack01Memo(val, wt, W - wt[n - 1], n - 1, dp),
                knapsack01Memo(val, wt, W, n - 1, dp));
        return dp[n][W];
    }

    //Approach: Dynamic Programming
    Codeium: Refactor | Explain | ×
    public static int knapsack01DP(int[] val, int[] wt, int W) {
        int n = val.length;
        int[][] dp = new int[n + 1][W + 1];

        //Build table dp[][] in bottom up manner
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= W ; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                } else if (wt[i - 1] <= j) {
                    //Return the maximum of two cases:
                    dp[i][j] = Math.max(val[i - 1] + dp[i - 1][j - wt[i - 1]], dp[i - 1][j]);
                } else {
                    //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[n][W];
    }
```

```
 .Knaspsack01
knapsack01(val, wt, W) = 3
```

```java
Codeium: Refactor | Explain
public class SubSetSumProblem01 {
    //Given a set of non-negative integers, and a value sum, determine if there is a subset
    // of the given set with sum equal to given sum.
    //Example: set[] = {3, 34, 4, 12, 5, 2}, sum = 9 -> true
    //There is a subset (4, 5) with sum 9.
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        int[] set = {3,34,4,12,5,2};
        int sum = 9;
        System.out.println("isSubsetSum(set, sum) = " + isSubsetSum(set, sum));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static boolean isSubsetSum(int[] set, int sum) {
        int n = set.length;
        boolean[][] dp = new boolean[n + 1][sum + 1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true;
        }

        // If sum is not 0 and set is empty, then answer is false
        for (int i = 1; i <= sum; i++) {
            dp[0][i] = false;
        }

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum ; j++) {
                if (j < set[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    //Excluding the current element
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
                }
            }
        }

        return dp[n][sum];
    }
```

```
.SubSetSumProblem01
isSubsetSum(set, sum) = true
```

```java
Codeium: Refactor | Explain
public class EqualSumPartition02 {
    //The partition problem is to determine whether a given set can be partitioned
    //into two subsets such that the sum of elements in both subsets is the same.
    //Examples:
    //arr[] = {1, 5, 11, 5}
    //Output: true
    //The array can be partitioned as {1, 5, 5} and {11}
    //please use dynamic programming approach

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        int[] arr = {1, 5, 11, 5};
        System.out.println("isEqualSumPartition(arr) = " + isEqualSumPartition(arr));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static boolean isEqualSumPartition(int[] arr) {
        int sum = 0;
        for (int i : arr) {
            sum += i;
        }
        if (sum % 2 != 0) {
            return false;
        }
        sum = sum / 2;
        int n = arr.length;
        boolean[][] dp = new boolean[n + 1][sum + 1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true;
        }

        // If sum is not 0 and set is empty, then answer is false
        for (int i = 1; i <= sum; i++) {
            dp[0][i] = false;
        }

```

```java

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum ; j++) {
                if (j < arr[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    //Excluding the current element
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - arr[i - 1]];
                }
            }
        }

        return dp[n][sum];
    }

```

```
.EqualSumPartition02
isEqualSumPartition(arr) = true
```

```java
Codeium: Refactor | Explain
public class CountOfSubsetOfGivenSum03 {
    //Given a set of non-negative integers, and a value sum, determine the number of subset
    // of the given set with sum equal to given sum.
    //Example: set[] = {3, 34, 4, 12, 5, 2}, sum = 9 -> 2
    //There are 2 subsets (4, 5) and (3, 4, 2) with sum 9.
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        int[] set = {3, 34, 4, 12, 5, 2};
        int sum = 9;
        System.out.println("countOfSubsetSum(set, sum) = " + countOfSubsetSum(set, sum));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int countOfSubsetSum(int[] set, int sum) {
        int n = set.length;
        int[][] dp = new int[n + 1][sum + 1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++) {
            dp[i][0] = 1;
        }

        // If sum is not 0 and set is empty, then answer is false
        for (int i = 1; i <= sum; i++) {
            dp[0][i] = 0;
        }

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum ; j++) {
                if (j < set[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    //Excluding the current element
                    dp[i][j] = dp[i - 1][j] + dp[i - 1][j - set[i - 1]];
                }
            }
        }

        return dp[n][sum];
    }

```

```
.CountOfSubsetOfGivenSum03
countOfSubsetSum(set, sum) = 2
```

```java
Codeium: Refactor | Explain
public class MinimumSubsetSumDiifference04 {
    //Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference
    // between their sums is minimum.
    //If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements
    // and the value of abs(sum(Subset1) - sum(Subset2)) should be minimum.
    //Example: set[] = {3, 1, 4, 2, 2, 1}, sum = 9 -> 1
    //The minimum difference between sum of two subsets is 1
    //
    // {1, 3, 4} & {1, 2, 2}
    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        //int[] set = {3, 1, 4, 2, 2, 1};
        int[] set = {1, 6, 11, 5};
        System.out.println("minimumSubsetSumDifference(set) = " + minimumSubsetSumDifference(set));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int minimumSubsetSumDifference(int[] set) {
        int n = set.length;
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += set[i];
        }

        boolean[][] dp = new boolean[n + 1][sum + 1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true;
        }

        // If sum is not 0 and set is empty, then answer is false
        for (int i = 1; i <= sum; i++) {
            dp[0][i] = false;
        }

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum ; j++) {
                if (j < set[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    //Excluding the current element
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
                }
            }
        }

        int diff = Integer.MAX_VALUE;
        for (int j = sum / 2; j >= 0; j--) {
            if (dp[n][j]) {
                diff = sum - 2 * j;
                break;
            }
        }

        return diff;
    }
}
```

```
.MinimumSubsetSumDiifference04
minimumSubsetSumDifference(set) = 1
```

```java
public class NoOfSubsetGivenDifference05 {
  //Given an array arr[] of size N and a given difference diff, the task is to count the number of partitions that
    // we can perform such that the difference between the sum of the two subsets is equal to the given difference.
    //Example: arr[] = {1, 1, 2, 3}, diff = 1 ->
    //possible partitions are {1, 1, 2} & {3}, {1, 3} & {1, 2}, {1, 1, 3} & {2}
    //Approach: Dynamic Programming
    //The problem can be reduced to count of subsets sum with a given sum. The idea is to consider the last element
    // in every subset first and then recur for the remaining array elements with the sum equal to the difference of
    // the total sum of the two subsets and the current element.
    //The base cases of the above recursive approach will be:
    //1. If the sum is 0, then return 1 (Empty subset allowed)
    //2. If the sum is not 0 and the index is 0, then return 0.
    //3. If the last element is greater than the sum, then ignore it.
//4. Else, we can either include it in the subset or exclude it from the subset.

    Codeium: Refactor | Explain | Generate Javadoc | X
    public static void main(String[] args) {
        int[] set = {1, 1, 2, 3};
        int diff = 1;
        System.out.println("noOfSubsetGivenDifference(set, diff) = " + noOfSubsetGivenDifference(set, diff));
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public static int noOfSubsetGivenDifference(int[] set, int diff) {
        int n = set.length;
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += set[i];
        }

        int s1 = (diff + sum) / 2;
        return countOfSubsetSum(set, s1);
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public static int countOfSubsetSum(int[] set, int sum) {
        int n = set.length;
        int[][] dp = new int[n + 1][sum + 1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++) {
            dp[i][0] = 1;
        }

        // If sum is not 0 and set is empty, then answer is false
        for (int i = 1; i <= sum; i++) {
            dp[0][i] = 0;
        }

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum ; j++) {
                if (j < set[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    //Excluding the current element
                    dp[i][j] = dp[i - 1][j] + dp[i - 1][j - set[i - 1]];
                }
            }
        }

        return dp[n][sum];
    }
}
```

```
.NoOfSubsetGivenDifference05
noOfSubsetGivenDifference(set, diff) = 3
```

Codeium: Refactor | Explain

```java
public class TargetSum06 {
    //Given an array arr[] of length N and an integer target. You want to build an expression out of arr[]
    // by adding one of the symbols '+' and '-' before each integer in arr[] and then
    // concatenate all the integers. Return the number of different expressions that can be built, which evaluates to target.
    //Example: arr[] = {1, 1, 1, 1, 1}, target = 3 -> 5
    //Input : N = 5, arr[] = {1, 1, 1, 1, 1}, target = 3
    //Output: 5
    //Explanation:
    //There are 5 ways to assign symbols to
    //make the sum of array be target 3.
    //
    //-1 + 1 + 1 + 1 + 1 = 3
    //+1 - 1 + 1 + 1 + 1 = 3
    //+1 + 1 - 1 + 1 + 1 = 3
    //+1 + 1 + 1 - 1 + 1 = 3
    //+1 + 1 + 1 + 1 - 1 = 3

    //Approach: Dynamic Programming

    // Codeium: Refactor | Explain | Generate Javadoc | X
    public static void main(String[] args) {
        int[] arr = {1, 1, 1, 1, 1};
        int target = 3;
        System.out.println("targetSum(arr, target) = " + targetSum(arr, target));

    }

```

Codeium: Refactor | Explain | Generate Javadoc | X

```java
    public static int targetSum(int[] nums, int target) {
        int sum = 0;
        for(int x : nums)
            sum += x;
        if(((sum - target) % 2 == 1) || (target > sum))
            return 0;

        int n = nums.length;
        int s2 = (sum - target)/2;
        int[][] t = new int[n + 1][s2 + 1];
        t[0][0] = 1;

        for(int i = 1; i < n + 1; i++) {
            for(int j = 0; j < s2 + 1; j++) {
                if(nums[i - 1] <= j)
                    t[i][j] = t[i-1][j] + t[i - 1][j - nums[i - 1]];
                else
                    t[i][j] = t[i - 1][j];
            }
        }
        return t[n][s2];
    }

}
```

```
.TargetSum06
targetSum(arr, target) = 5
```

```java
Codeium: Refactor | Explain
public class unboundedKnapsack {
    //Given a knapsack weight W and a set of n items with certain value vali and weight wti,
    // we need to calculate the maximum amount that could make up this quantity exactly.
    // This is different from classical Knapsack problem, here we are allowed to use unlimited number of instances of an item.
    //Example: W = 100; val[] = {10, 30, 20}; wt[] = {5, 10, 15} -> 300
    //Approach: Recursion

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        int[] val = {10, 30, 20};
        int[] wt = {5, 10, 15};
        int W = 100;
        System.out.println("unboundedKnapsack(val, wt, W) = " + unboundedKnapsack(val, wt, W));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int unboundedKnapsack(int[] val, int[] wt, int W) {
        int n = val.length;
        return unboundedKnapsack(val, wt, W, n);
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int unboundedKnapsack(int[] val, int[] wt, int W, int n) {
        //Base condition
        if (n == 0 || W == 0) {
            return 0;
        }

        //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
        if (wt[n - 1] > W) {
            return unboundedKnapsack(val, wt, W, n - 1);
        }

        //Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        return Math.max(val[n - 1] + unboundedKnapsack(val, wt, W - wt[n - 1], n), unboundedKnapsack(val, wt, W, n - 1));

    }
```

```java
    //Approach: Memoization
    Codeium: Refactor | Explain | ×
    public static int unboundedKnapsackMemo(int[] val, int[] wt, int W) {
        int n = val.length;
        int[][] dp = new int[n + 1][W + 1];
        return unboundedKnapsackMemo(val, wt, W, n, dp);
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int unboundedKnapsackMemo(int[] val, int[] wt, int W, int n, int[][] dp) {
        //Base condition
        if (n == 0 || W == 0) {
            return 0;
        }

        if (dp[n][W] != 0) {
            return dp[n][W];
        }

        //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
        if (wt[n - 1] > W) {
            dp[n][W] = unboundedKnapsackMemo(val, wt, W, n - 1, dp);
            return dp[n][W];
        }

        //Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        dp[n][W] = Math.max(val[n - 1] + unboundedKnapsackMemo(val, wt, W - wt[n - 1], n, dp),
                unboundedKnapsackMemo(val, wt, W, n - 1, dp));
        return dp[n][W];
    }

```

```java
    //Approach: Dynamic Programming
    Codeium: Refactor | Explain | ×
    public static int unboundedKnapsackDP(int[] val, int[] wt, int W) {
        int n = val.length;
        int[][] dp = new int[n + 1][W + 1];
        // Build table dp[][] in bottom up manner
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= W ; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                } else if (wt[i - 1] <= j) {
                    dp[i][j] = Math.max(val[i - 1] + dp[i][j - wt[i - 1]], dp[i - 1][j]);
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        return dp[n][W];
    }

    //Approach: Dynamic Programming (Space Optimized)
    Codeium: Refactor | Explain | ×
    public static int unboundedKnapsackDPSpaceOptimized(int[] val, int[] wt, int W) {
        int n = val.length;
        int[] dp = new int[W + 1];

        // Build table dp[][] in bottom up manner
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= W ; j++) {
                if (i == 0 || j == 0) {
                    dp[j] = 0;
                } else if (wt[i - 1] <= j) {
                    dp[j] = Math.max(val[i - 1] + dp[j - wt[i - 1]], dp[j]);
                } else {
                    dp[j] = dp[j];
                }

            }
        }
        return dp[W];
    }
```

```
 .unboundedKnapsack
unboundedKnapsack(val, wt, W) = 300
```

```java
Codeium: Refactor | Explain
public class RodCuttingProblemn01 {
    //Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n.
    // Determine the maximum value obtainable by cutting up the rod and selling the pieces.
    //Example: length[] = {1, 2, 3, 4, 5, 6, 7, 8}; price[] = {1, 5, 8, 9, 10, 17, 17, 20}; n = 8 -> 22

    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | X
    public static void main(String[] args) {
        int[] length = {1, 2, 3, 4, 5, 6, 7, 8};
        int[] price = {1, 5, 8, 9, 10, 17, 17, 20};
        int n = 8;
        System.out.println("rodCuttingProblem(length, price, n) = " + rodCuttingProblem(length, price, n));
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public static int rodCuttingProblem(int[] length, int[] price, int n) {
        int[][] dp = new int[length.length + 1][n + 1];

        // If length is 0, then answer is 0
        for (int i = 0; i <= length.length; i++) {
            dp[i][0] = 0;
        }

        // If length is not 0 and price is 0, then answer is 0
        for (int i = 1; i <= n; i++) {
            dp[0][i] = 0;
        }

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= length.length; i++) {
            for (int j = 1; j <= n ; j++) {
                if (j < length[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    //Excluding the current element
                    dp[i][j] = Math.max(dp[i - 1][j], price[i - 1] + dp[i][j - length[i - 1]]);
                }
            }
        }

        return dp[length.length][n];
    }
}
```

```
.RodCuttingProblemn01
rodCuttingProblem(length, price, n) = 22
```

```java
Codeium: Refactor | Explain
public class MaximumCoinChange {
    //Given a value N, if we want to make change for N cents, and we have infinite supply of each of
    // S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn’t matter.
    //Example: N = 4; S = {1, 2, 3} -> 4
    //1 + 1 + 1 + 1     1 + 1 + 2     2 + 2     1 + 3

    //Approach: Dynamic Programming

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static void main(String[] args) {
        int[] coins = {1, 2, 3};
        int n = 4;
        System.out.println("maximumCoinChange(coins, n) = " + maximumCoinChange(coins, n));
    }

    Codeium: Refactor | Explain | Generate Javadoc | ×
    public static int maximumCoinChange(int[] coins, int n) {
        int[][] dp = new int[coins.length + 1][n + 1];

        // If n is 0, then answer is 1
        for (int i = 0; i <= coins.length; i++) {
            dp[i][0] = 1;
        }

        // If n is not 0 and coins is 0, then answer is 0
        for (int i = 1; i <= n; i++) {
            dp[0][i] = 0;
        }

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= coins.length; i++) {
            for (int j = 1; j <= n ; j++) {
                if (j < coins[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    //Excluding the current element
                    dp[i][j] = dp[i - 1][j] + dp[i][j - coins[i - 1]];
                }
            }
        }

        return dp[coins.length][n];
    }
```

```
.MaximumCoinChange
maximumCoinChange(coins, n) = 4
```

```java
Codeium: Refactor | Explain
public class MinimumCoinChange {
    //Given a value N, if we want to make change for N cents, and we have infinite supply of each of
    // S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.
    //Example: N = 4; S = {1, 2, 3} -> 4
    //1 + 1 + 1 + 1     1 + 1 + 2     2 + 2     1 + 3
    //Approach: Dynamic Programming

```

```java
    Codeium: Refactor | Explain | Generate Javadoc | X
    public static void main(String[] args) {
//        int[] coins = {1, 2, 3};
//        int n = 4;
//        int[] coins = {25, 10, 5};
//        int n = 30;
        int[] coins = {9,6,5,1};
        int n = 11;
        System.out.println("minimumCoinChange(coins, n) = " + minimumCoinChange(coins, n));
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public static int minimumCoinChange(int[] coins, int n) {
        int[][] dp = new int[coins.length + 1][n + 1];

        // If n is 0, then answer is 1
        for (int i = 0; i <= coins.length; i++) {
            dp[i][0] = 0;
        }

        // If n is not 0 and coins is 0, then answer is 0
        for (int i = 1; i <= n; i++) {
            dp[0][i] = Integer.MAX_VALUE - 1;
        }

        // Fill the subset table in bottom up manner
        for (int i = 1; i <= coins.length; i++) {
            for (int j = 1; j <= n ; j++) {
                if (j < coins[i - 1]) {
                    dp[i][j] = dp[i - 1][j];
                } else {
                    //Excluding the current element
                    dp[i][j] = Math.min(dp[i - 1][j], 1 + dp[i][j - coins[i - 1]]);
                }
            }
        }

        return dp[coins.length][n];
    }
}

```

```
.MinimumCoinChange
minimumCoinChange(coins, n) = 2
```