

Codeium: Refactor | Explain

```
3 public class Knapsack01 {
4     //Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value
5     // in the knapsack.
6     //In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and
7     //weights associated with n items
8     //respectively. Also given an integer W which represents knapsack capacity, find out the
9     //maximum value subset of val[] such that
10    //sum of the weights of this subset is smaller than or equal to W. You cannot break
11    //an item, either pick the complete item, or
12    //don't pick it (0-1 property).
13    //Example: val[] = {60, 100, 120}; wt[] = {10, 20, 30}; W = 50 -> 220
14    //Approach: Recursion
15
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
16    public static void main(String[] args) {
17        //        int[] val = {60, 100, 120};
18        //        int[] wt = {10, 20, 30};
19        //        int W = 50;
20        int[] val = {1, 2, 3};
21        int[] wt = {4, 5, 1};
22        int W = 4;
23        System.out.println("knapsack01(val, wt, W) = " + knapsack01(val, wt, W));
24    }
25
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
26    public static int knapsack01(int[] val, int[] wt, int W) {
27        int n = val.length;
28        return knapsack01(val, wt, W, n);
29    }
30
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
31    public static int knapsack01(int[] val, int[] wt, int W, int n) {
32        //Base condition
33        if (n == 0 || W == 0) {
34            return 0;
35        }
36
37        //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
38        if (wt[n - 1] > W) {
39            return knapsack01(val, wt, W, n - 1);
40        }
41
42        //Return the maximum of two cases:
43        // (1) nth item included
44        // (2) not included
45        return Math.max(val[n - 1] + knapsack01(val, wt, W - wt[n - 1], n - 1), knapsack01(val, wt, W, n - 1));
46    }
47
```

```

48 //Approach: Memoization
49 Codeium: Refactor | Explain | ✕
50 public static int knapsack01Memo(int[] val, int[] wt, int W) {
51     int n = val.length;
52     int[][] dp = new int[n + 1][W + 1];
53     return knapsack01Memo(val, wt, W, n, dp);
54 }
55 Codeium: Refactor | Explain | Generate Javadoc | ✕
56 public static int knapsack01Memo(int[] val, int[] wt, int W, int n, int[][] dp) {
57     //Base condition
58     if (n == 0 || W == 0) {
59         return 0;
60     }
61     if (dp[n][W] != 0) {
62         return dp[n][W];
63     }
64     //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
65     if (wt[n - 1] > W) {
66         dp[n][W] = knapsack01Memo(val, wt, W, n - 1, dp);
67         return dp[n][W];
68     }
69     //Return the maximum of two cases:
70     // (1) nth item included
71     // (2) not included
72     dp[n][W] = Math.max(val[n - 1] + knapsack01Memo(val, wt, W - wt[n - 1], n - 1, dp),
73         knapsack01Memo(val, wt, W, n - 1, dp));
74     return dp[n][W];
75 }
76
77
78
79 //Approach: Dynamic Programming
80 Codeium: Refactor | Explain | ✕
81 public static int knapsack01DP(int[] val, int[] wt, int W) {
82     int n = val.length;
83     int[][] dp = new int[n + 1][W + 1];
84     //Build table dp[][] in bottom up manner
85     for (int i = 0; i <= n; i++) {
86         for (int j = 0; j <= W; j++) {
87             if (i == 0 || j == 0) {
88                 dp[i][j] = 0;
89             } else if (wt[i - 1] <= j) {
90                 //Return the maximum of two cases:
91                 dp[i][j] = Math.max(val[i - 1] + dp[i - 1][j - wt[i - 1]], dp[i - 1][j]);
92             } else {
93                 //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
94                 dp[i][j] = dp[i - 1][j];
95             }
96         }
97     }
98     return dp[n][W];
99 }
100

```

## .Knapsack01

knapsack01(val, wt, W) = 3

Codeium: Refactor | Explain

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

Codeium: Refactor | Explain | Generate Javadoc | ✕

```
10 public static void main(String[] args) {
11     int[] set = {3,34,4,12,5,2};
12     int sum = 9;
13     System.out.println("isSubsetSum(set, sum) = " + isSubsetSum(set, sum));
14 }
15
```

Codeium: Refactor | Explain | Generate Javadoc | ✕

```
16 public static boolean isSubsetSum(int[] set, int sum) {
17     int n = set.length;
18     boolean[][] dp = new boolean[n + 1][sum + 1];
19
20     // If sum is 0, then answer is true
21     for (int i = 0; i <= n; i++) {
22         dp[i][0] = true;
23     }
24
25     // If sum is not 0 and set is empty, then answer is false
26     for (int i = 1; i <= sum; i++) {
27         dp[0][i] = false;
28     }
29
30     // Fill the subset table in bottom up manner
31     for (int i = 1; i <= n; i++) {
32         for (int j = 1; j <= sum ; j++) {
33             if (j < set[i - 1]) {
34                 dp[i][j] = dp[i - 1][j];
35             } else {
36                 //Excluding the current element
37                 dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
38             }
39         }
40     }
41
42     return dp[n][sum];
43 }
```

**.SubSetSumProblem01**

**isSubsetSum(set, sum) = true**

```

-
Codeium: Refactor | Explain
3 public class EqualSumPartition02 {
4     //The partition problem is to determine whether a given set can be partitioned
5     //into two subsets such that the sum of elements in both subsets is the same.
6     //Examples:
7     //arr[] = {1, 5, 11, 5}
8     //Output: true
9     //The array can be partitioned as {1, 5, 5} and {11}
10    //please use dynamic programming approach
11
Codeium: Refactor | Explain | Generate Javadoc | ✕
12 public static void main(String[] args) {
13     int[] arr = {1, 5, 11, 5};
14     System.out.println("isEqualSumPartition(arr) = " + isEqualSumPartition(arr));
15 }
16
Codeium: Refactor | Explain | Generate Javadoc | ✕
17 public static boolean isEqualSumPartition(int[] arr) {
18     int sum = 0;
19     for (int i : arr) {
20         sum += i;
21     }
22     if (sum % 2 != 0) {
23         return false;
24     }
25     sum = sum / 2;
26     int n = arr.length;
27     boolean[][] dp = new boolean[n + 1][sum + 1];
28
29     // If sum is 0, then answer is true
30     for (int i = 0; i <= n; i++) {
31         dp[i][0] = true;
32     }
33
34     // If sum is not 0 and set is empty, then answer is false
35     for (int i = 1; i <= sum; i++) {
36         dp[0][i] = false;
37     }
38
39     // Fill the subset table in bottom up manner
40     for (int i = 1; i <= n; i++) {
41         for (int j = 1; j <= sum; j++) {
42             if (j < arr[i - 1]) {
43                 dp[i][j] = dp[i - 1][j];
44             } else {
45                 //Excluding the current element
46                 dp[i][j] = dp[i - 1][j] || dp[i - 1][j - arr[i - 1]];
47             }
48         }
49     }
50
51     return dp[n][sum];
52 }
53
-

```

**.EqualSumPartition02**

**isEqualSumPartition(arr) = true**

Codeium: Refactor | Explain

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

Codeium: Refactor | Explain | Generate Javadoc | X

```
10 public static void main(String[] args) {
11     int[] set = {3, 34, 4, 12, 5, 2};
12     int sum = 9;
13     System.out.println("countOfSubsetSum(set, sum) = " + countOfSubsetSum(set, sum));
14 }
15
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
16 public static int countOfSubsetSum(int[] set, int sum) {
17     int n = set.length;
18     int[][] dp = new int[n + 1][sum + 1];
19
20     // If sum is 0, then answer is true
21     for (int i = 0; i <= n; i++) {
22         dp[i][0] = 1;
23     }
24
25     // If sum is not 0 and set is empty, then answer is false
26     for (int i = 1; i <= sum; i++) {
27         dp[0][i] = 0;
28     }
29
30     // Fill the subset table in bottom up manner
31     for (int i = 1; i <= n; i++) {
32         for (int j = 1; j <= sum; j++) {
33             if (j < set[i - 1]) {
34                 dp[i][j] = dp[i - 1][j];
35             } else {
36                 //Excluding the current element
37                 dp[i][j] = dp[i - 1][j] + dp[i - 1][j - set[i - 1]];
38             }
39         }
40     }
41
42     return dp[n][sum];
43 }
44
```

.CountOfSubsetOfGivenSum03

countOfSubsetSum(set, sum) = 2

```

Codeium: Refactor | Explain
3 public class MinimumSubsetSumDifference04 {
4     //Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference
5     // between their sums is minimum.
6     //If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements
7     // and the value of abs(sum(Subset1) - sum(Subset2)) should be minimum.
8     //Example: set[] = {3, 1, 4, 2, 2, 1}, sum = 9 -> 1
9     //The minimum difference between sum of two subsets is 1
10    //
11    // {1, 3, 4} & {1, 2, 2}
12    //Approach: Dynamic Programming
13
Codeium: Refactor | Explain | Generate Javadoc | X
14 public static void main(String[] args) {
15     //int[] set = {3, 1, 4, 2, 2, 1};
16     int[] set = {1, 6, 11, 5};
17     System.out.println("minimumSubsetSumDifference(set) = " + minimumSubsetSumDifference(set));
18 }
19

```

```

Codeium: Refactor | Explain | Generate Javadoc | X
20 public static int minimumSubsetSumDifference(int[] set) {
21     int n = set.length;
22     int sum = 0;
23     for (int i = 0; i < n; i++) {
24         sum += set[i];
25     }
26
27     boolean[][] dp = new boolean[n + 1][sum + 1];
28
29     // If sum is 0, then answer is true
30     for (int i = 0; i <= n; i++) {
31         dp[i][0] = true;
32     }
33
34     // If sum is not 0 and set is empty, then answer is false
35     for (int i = 1; i <= sum; i++) {
36         dp[0][i] = false;
37     }
38
39     // Fill the subset table in bottom up manner
40     for (int i = 1; i <= n; i++) {
41         for (int j = 1; j <= sum; j++) {
42             if (j < set[i - 1]) {
43                 dp[i][j] = dp[i - 1][j];
44             } else {
45                 //Excluding the current element
46                 dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
47             }
48         }
49     }
50
51     int diff = Integer.MAX_VALUE;
52     for (int j = sum / 2; j >= 0; j--) {
53         if (dp[n][j]) {
54             diff = sum - 2 * j;
55             break;
56         }
57     }
58
59     return diff;
60 }
61 }
62

```

**.MinimumSubsetSumDifference04**

**minimumSubsetSumDifference(set) = 1**

```

3 public class NoOfSubsetGivenDifference05 {
4     //Given an array arr[] of size N and a given difference diff, the task is to count the number of partitions that
5     // we can perform such that the difference between the sum of the two subsets is equal to the given difference.
6     //Example: arr[] = {1, 1, 2, 3}, diff = 1 ->
7     //possible partitions are {1, 1, 2} & {3}, {1, 3} & {1, 2}, {1, 1, 3} & {2}
8     //Approach: Dynamic Programming
9     //The problem can be reduced to count of subsets sum with a given sum. The idea is to consider the last element
10    // in every subset first and then recur for the remaining array elements with the sum equal to the difference of
11    // the total sum of the two subsets and the current element.
12    //The base cases of the above recursive approach will be:
13    //1. If the sum is 0, then return 1 (Empty subset allowed)
14    //2. If the sum is not 0 and the index is 0, then return 0.
15    //3. If the last element is greater than the sum, then ignore it.
16    //4. Else, we can either include it in the subset or exclude it from the subset.
17
18    Codeium: Refactor | Explain | Generate Javadoc | X
19    public static void main(String[] args) {
20        int[] set = {1, 1, 2, 3};
21        int diff = 1;
22        System.out.println("noOfSubsetGivenDifference(set, diff) = " + noOfSubsetGivenDifference(set, diff));
23    }

```

```

24    Codeium: Refactor | Explain | Generate Javadoc | X
25    public static int noOfSubsetGivenDifference(int[] set, int diff) {
26        int n = set.length;
27        int sum = 0;
28        for (int i = 0; i < n; i++) {
29            sum += set[i];
30        }
31        int s1 = (diff + sum) / 2;
32        return countOfSubsetSum(set, s1);
33    }
34

```

```

35    Codeium: Refactor | Explain | Generate Javadoc | X
36    public static int countOfSubsetSum(int[] set, int sum) {
37        int n = set.length;
38        int[][] dp = new int[n + 1][sum + 1];
39
40        // If sum is 0, then answer is true
41        for (int i = 0; i <= n; i++) {
42            dp[i][0] = 1;
43        }
44
45        // If sum is not 0 and set is empty, then answer is false
46        for (int i = 1; i <= sum; i++) {
47            dp[0][i] = 0;
48        }
49
50        // Fill the subset table in bottom up manner
51        for (int i = 1; i <= n; i++) {
52            for (int j = 1; j <= sum; j++) {
53                if (j < set[i - 1]) {
54                    dp[i][j] = dp[i - 1][j];
55                } else {
56                    //Excluding the current element
57                    dp[i][j] = dp[i - 1][j] + dp[i - 1][j - set[i - 1]];
58                }
59            }
60        }
61        return dp[n][sum];
62    }
63 }
64

```

**.NoOfSubsetGivenDifference05**

**noOfSubsetGivenDifference(set, diff) = 3**

```

Codeium: Refactor | Explain
3 public class TargetSum06 {
4     //Given an array arr[] of length N and an integer target. You want to build an expression out of arr[]
5     // by adding one of the symbols '+' and '-' before each integer in arr[] and then
6     // concatenate all the integers. Return the number of different expressions that can be built, which evaluates to target.
7     //Example: arr[] = {1, 1, 1, 1, 1}, target = 3 -> 5
8     //Input : N = 5, arr[] = {1, 1, 1, 1, 1}, target = 3
9     //Output: 5
10    //Explanation:
11    //There are 5 ways to assign symbols to
12    //make the sum of array be target 3.
13    //
14    //-1 + 1 + 1 + 1 + 1 = 3
15    //+1 - 1 + 1 + 1 + 1 = 3
16    //+1 + 1 - 1 + 1 + 1 = 3
17    //+1 + 1 + 1 - 1 + 1 = 3
18    //+1 + 1 + 1 + 1 - 1 = 3
19
20    //Approach: Dynamic Programming
21
Codeium: Refactor | Explain | Generate Javadoc | X
22 public static void main(String[] args) {
23     int[] arr = {1, 1, 1, 1, 1};
24     int target = 3;
25     System.out.println("targetSum(arr, target) = " + targetSum(arr, target));
26
27 }
28

```

```

Codeium: Refactor | Explain | Generate Javadoc | X
31 public static int targetSum(int[] nums, int target) {
32     int sum = 0;
33     for(int x : nums)
34         sum += x;
35     if(((sum - target) % 2 == 1) || (target > sum))
36         return 0;
37
38     int n = nums.length;
39     int s2 = (sum - target)/2;
40     int[][] t = new int[n + 1][s2 + 1];
41     t[0][0] = 1;
42
43     for(int i = 1; i < n + 1; i++) {
44         for(int j = 0; j < s2 + 1; j++) {
45             if(nums[i - 1] <= j)
46                 t[i][j] = t[i-1][j] + t[i - 1][j - nums[i - 1]];
47             else
48                 t[i][j] = t[i - 1][j];
49         }
50     }
51     return t[n][s2];
52 }
53
54 }
55
56

```

**.TargetSum06**

targetSum(arr, target) = 5



```

Codeium: Refactor | Explain
3 public class unboundedKnapsack {
4     //Given a knapsack weight W and a set of n items with certain value val[] and weight wt[],
5     // we need to calculate the maximum amount that could make up this quantity exactly.
6     // This is different from classical Knapsack problem, here we are allowed to use unlimited number of instances of an item.
7     //Example: W = 100; val[] = {10, 30, 20}; wt[] = {5, 10, 15} -> 300
8     //Approach: Recursion
9
Codeium: Refactor | Explain | Generate Javadoc | ✕
10 public static void main(String[] args) {
11     int[] val = {10, 30, 20};
12     int[] wt = {5, 10, 15};
13     int W = 100;
14     System.out.println("unboundedKnapsack(val, wt, W) = " + unboundedKnapsack(val, wt, W));
15 }
16
Codeium: Refactor | Explain | Generate Javadoc | ✕
17 public static int unboundedKnapsack(int[] val, int[] wt, int W) {
18     int n = val.length;
19     return unboundedKnapsack(val, wt, W, n);
20 }
21
Codeium: Refactor | Explain | Generate Javadoc | ✕
22 public static int unboundedKnapsack(int[] val, int[] wt, int W, int n) {
23     //Base condition
24     if (n == 0 || W == 0) {
25         return 0;
26     }
27
28     //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
29     if (wt[n - 1] > W) {
30         return unboundedKnapsack(val, wt, W, n - 1);
31     }
32
33     //Return the maximum of two cases:
34     // (1) nth item included
35     // (2) not included
36     return Math.max(val[n - 1] + unboundedKnapsack(val, wt, W - wt[n - 1], n), unboundedKnapsack(val, wt, W, n - 1));
37 }
38
40 //Approach: Memoization
Codeium: Refactor | Explain | ✕
41 public static int unboundedKnapsackMemo(int[] val, int[] wt, int W) {
42     int n = val.length;
43     int[][] dp = new int[n + 1][W + 1];
44     return unboundedKnapsackMemo(val, wt, W, n, dp);
45 }
46
Codeium: Refactor | Explain | Generate Javadoc | ✕
47 public static int unboundedKnapsackMemo(int[] val, int[] wt, int W, int n, int[][] dp) {
48     //Base condition
49     if (n == 0 || W == 0) {
50         return 0;
51     }
52
53     if (dp[n][W] != 0) {
54         return dp[n][W];
55     }
56
57     //If weight of the nth item is more than Knapsack capacity W, then this item cannot be included in the optimal solution
58     if (wt[n - 1] > W) {
59         dp[n][W] = unboundedKnapsackMemo(val, wt, W, n - 1, dp);
60         return dp[n][W];
61     }
62
63     //Return the maximum of two cases:
64     // (1) nth item included
65     // (2) not included
66     dp[n][W] = Math.max(val[n - 1] + unboundedKnapsackMemo(val, wt, W - wt[n - 1], n, dp),
67         unboundedKnapsackMemo(val, wt, W, n - 1, dp));
68     return dp[n][W];
69 }
70

```

```

71 //Approach: Dynamic Programming
72 Codeium: Refactor | Explain | X
73 public static int unboundedKnapsackDP(int[] val, int[] wt, int W) {
74     int n = val.length;
75     int[][] dp = new int[n + 1][W + 1];
76     // Build table dp[][] in bottom up manner
77     for (int i = 0; i <= n; i++) {
78         for (int j = 0; j <= W; j++) {
79             if (i == 0 || j == 0) {
80                 dp[i][j] = 0;
81             } else if (wt[i - 1] <= j) {
82                 dp[i][j] = Math.max(val[i - 1] + dp[i][j - wt[i - 1]], dp[i - 1][j]);
83             } else {
84                 dp[i][j] = dp[i - 1][j];
85             }
86         }
87     }
88     return dp[n][W];
89 }
90
91 //Approach: Dynamic Programming (Space Optimized)
92 Codeium: Refactor | Explain | X
93 public static int unboundedKnapsackDPSpaceOptimized(int[] val, int[] wt, int W) {
94     int n = val.length;
95     int[] dp = new int[W + 1];
96
97     // Build table dp[][] in bottom up manner
98     for (int i = 0; i <= n; i++) {
99         for (int j = 0; j <= W; j++) {
100             if (i == 0 || j == 0) {
101                 dp[j] = 0;
102             } else if (wt[i - 1] <= j) {
103                 dp[j] = Math.max(val[i - 1] + dp[j - wt[i - 1]], dp[j]);
104             } else {
105                 dp[j] = dp[j];
106             }
107         }
108     }
109     return dp[W];
110 }

```

**.unboundedKnapsack**

**unboundedKnapsack(val, wt, W) = 300**

```

2 Codeium: Refactor | Explain
3 public class RodCuttingProblemn01 {
4     //Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n.
5     // Determine the maximum value obtainable by cutting up the rod and selling the pieces.
6     //Example: length[] = {1, 2, 3, 4, 5, 6, 7, 8}; price[] = {1, 5, 8, 9, 10, 17, 17, 20}; n = 8 -> 22
7     //Approach: Dynamic Programming
8
9     Codeium: Refactor | Explain | Generate Javadoc | X
10    public static void main(String[] args) {
11        int[] length = {1, 2, 3, 4, 5, 6, 7, 8};
12        int[] price = {1, 5, 8, 9, 10, 17, 17, 20};
13        int n = 8;
14        System.out.println("rodCuttingProblem(length, price, n) = " + rodCuttingProblem(length, price, n));
15    }
16
17    Codeium: Refactor | Explain | Generate Javadoc | X
18    public static int rodCuttingProblem(int[] length, int[] price, int n) {
19        int[][] dp = new int[length.length + 1][n + 1];
20
21        // If length is 0, then answer is 0
22        for (int i = 0; i <= length.length; i++) {
23            dp[i][0] = 0;
24        }
25
26        // If length is not 0 and price is 0, then answer is 0
27        for (int i = 1; i <= n; i++) {
28            dp[0][i] = 0;
29        }
30
31        // Fill the subset table in bottom up manner
32        for (int i = 1; i <= length.length; i++) {
33            for (int j = 1; j <= n; j++) {
34                if (j < length[i - 1]) {
35                    dp[i][j] = dp[i - 1][j];
36                } else {
37                    //Excluding the current element
38                    dp[i][j] = Math.max(dp[i - 1][j], price[i - 1] + dp[i][j - length[i - 1]]);
39                }
40            }
41        }
42
43        return dp[length.length][n];
44    }
45 }

```

**.RodCuttingProblemn01**

**rodCuttingProblem(length, price, n) = 22**

```

Codeium: Refactor | Explain
3 public class MaximumCoinChange {
4     //Given a value N, if we want to make change for N cents, and we have infinite supply of each of
5     // S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.
6     //Example: N = 4; S = {1, 2, 3} -> 4
7     //1 + 1 + 1 + 1    1 + 1 + 2    2 + 2    1 + 3
8
9     //Approach: Dynamic Programming
10
Codeium: Refactor | Explain | Generate Javadoc | X
11 public static void main(String[] args) {
12     int[] coins = {1, 2, 3};
13     int n = 4;
14     System.out.println("maximumCoinChange(coins, n) = " + maximumCoinChange(coins, n));
15 }
16
Codeium: Refactor | Explain | Generate Javadoc | X
17 public static int maximumCoinChange(int[] coins, int n) {
18     int[][] dp = new int[coins.length + 1][n + 1];
19
20     // If n is 0, then answer is 1
21     for (int i = 0; i <= coins.length; i++) {
22         dp[i][0] = 1;
23     }
24
25     // If n is not 0 and coins is 0, then answer is 0
26     for (int i = 1; i <= n; i++) {
27         dp[0][i] = 0;
28     }
29
30     // Fill the subset table in bottom up manner
31     for (int i = 1; i <= coins.length; i++) {
32         for (int j = 1; j <= n; j++) {
33             if (j < coins[i - 1]) {
34                 dp[i][j] = dp[i - 1][j];
35             } else {
36                 //Excluding the current element
37                 dp[i][j] = dp[i - 1][j] + dp[i][j - coins[i - 1]];
38             }
39         }
40     }
41
42     return dp[coins.length][n];
43 }

```

## `.MaximumCoinChange`

`maximumCoinChange(coins, n) = 4`

```

3 public class MinimumCoinChange {
4     //Given a value N, if we want to make change for N cents, and we have infinite supply of each of
5     // S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.
6     //Example: N = 4; S = {1, 2, 3} -> 4
7     //1 + 1 + 1 + 1    1 + 1 + 2    2 + 2    1 + 3
8     //Approach: Dynamic Programming
9

```

```

Codeium: Refactor | Explain | Generate Javadoc | X
10 public static void main(String[] args) {
11     //     int[] coins = {1, 2, 3};
12     //     int n = 4;
13     //     int[] coins = {25, 10, 5};
14     //     int n = 30;
15     int[] coins = {9,6,5,1};
16     int n = 11;
17     System.out.println("minimumCoinChange(coins, n) = " + minimumCoinChange(coins, n));
18 }
19
Codeium: Refactor | Explain | Generate Javadoc | X
20 public static int minimumCoinChange(int[] coins, int n) {
21     int[][] dp = new int[coins.length + 1][n + 1];
22
23     // If n is 0, then answer is 1
24     for (int i = 0; i <= coins.length; i++) {
25         dp[i][0] = 0;
26     }
27
28     // If n is not 0 and coins is 0, then answer is 0
29     for (int i = 1; i <= n; i++) {
30         dp[0][i] = Integer.MAX_VALUE - 1;
31     }
32
33     // Fill the subset table in bottom up manner
34     for (int i = 1; i <= coins.length; i++) {
35         for (int j = 1; j <= n ; j++) {
36             if (j < coins[i - 1]) {
37                 dp[i][j] = dp[i - 1][j];
38             } else {
39                 //Excluding the current element
40                 dp[i][j] = Math.min(dp[i - 1][j], 1 + dp[i][j - coins[i - 1]]);
41             }
42         }
43     }
44
45     return dp[coins.length][n];
46 }
47 }
48

```

## .MinimumCoinChange

minimumCoinChange(coins, n) = 2