**ntroduction**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx763A3x9Pl#introduction)

Given an infinite supply of ‘n’ coin denominations and a total money amount, we are asked to find the total number of distinct ways to make up that amount.

**Example:**

Denominations: {1,2,3}  
Total amount: 5  
Output: 5  
Explanation: There are five ways to make the change for '5', here are those ways:  
  1. {1,1,1,1,1}   
  2. {1,1,1,2}   
  3. {1,2,2}  
  4. {1,1,3}  
  5. {2,3}

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx763A3x9Pl#problem-statement)

Given a number array to represent different coin denominations and a total amount ‘T’, we need to find all the different ways to make a change for ‘T’ with the given coin denominations. We can assume an infinite supply of coins, therefore, each coin can be chosen multiple times.

This problem follows the [Unbounded Knapsack](https://www.educative.io/collection/page/5668639101419520/5633779737559040/5745865499082752/) pattern.

The time complexity of the recursive algorithm is exponential O(2^{C+T})*O*(2​*C*+*T*​​), where ‘C’ represents total coin denominations and ‘T’ is the total amount that we want to make change. The space complexity will be O(C+T)*O*(*C*+*T*).

Let’s try to find a better solution.

### Top-down Dynamic Programming with Memoization [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx763A3x9Pl#top-down-dynamic-programming-with-memoization)

We can use memoization to overcome the overlapping sub-problems. We will be using a two-dimensional array to store the results of solved sub-problems. As mentioned above, we need to store results for every coin combination and for every possible sum:

class CoinChange {

  public int countChange(int[] denominations, int total)

  {

    Integer[][] dp = new Integer[denominations.length][total + 1];

    return this.countChangeRecursive(dp, denominations, total, 0);

  }

  private int countChangeRecursive(Integer[][] dp, int[] denominations, int total, int currentIndex)

  {

    // base checks

    if (total == 0)

      return 1;

    if(denominations.length == 0 || currentIndex >= denominations.length)

      return 0;

    // if we have already processed a similar sub-problem, return the result from memory

    if(dp[currentIndex][total] != null)

      return dp[currentIndex][total];

    // recursive call after selecting the coin at the currentIndex

    // if the number at currentIndex exceeds the total, we shouldn't process this

    int sum1 = 0;

    if( denominations[currentIndex] <= total )

      sum1 = countChangeRecursive(dp, denominations, total - denominations[currentIndex], currentIndex);

    // recursive call after excluding the number at the currentIndex

    int sum2 = countChangeRecursive(dp, denominations, total, currentIndex + 1);

    dp[currentIndex][total] = sum1 + sum2;

    return dp[currentIndex][total];

  }

  public static void main(String[] args) {

    CoinChange cc = new CoinChange();

    int[] denominations = {1, 2, 3};

    System.out.println(cc.countChange(denominations, 5));

  }

}

### Bottom-up Dynamic Programming [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx763A3x9Pl#bottom-up-dynamic-programming)

We will try to find if we can make all possible sums, with every combination of coins, to populate the array dp[TotalDenominations][Total+1].

So for every possible total ‘t’ (0<= t <= Total) and for every possible coin index (0 <= index < denominations.length), we have two options:

1. Exclude the coin. Count all the coin combinations without the given coin up to the total ‘t’ => dp[index-1][t]
2. Include the coin if its value is not more than ‘t’. In this case, we will count all the coin combinations to get the remaining total: dp[index][t-denominations[index]]

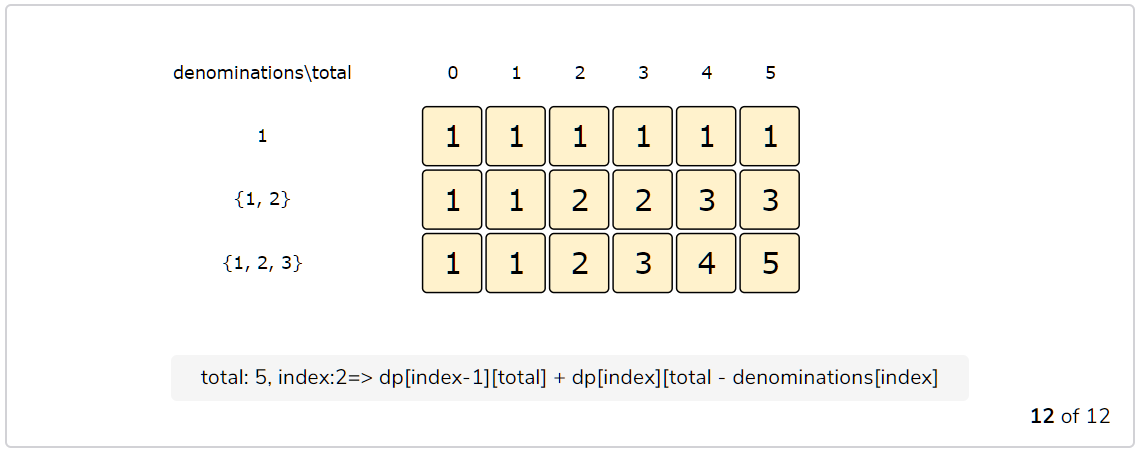
Finally, to find the total combinations, we will add both the above two values:

    dp[index][t] = dp[index-1][t] + dp[index][t-denominations[index]]

Let’s draw this visually with the following example:

    Denominations: [1, 2, 3]    
    Total: 5

Let’s start with our base case of zero total:



class CoinChange {

  public int countChange(int[] denominations, int total) {

    int n = denominations.length;

    int[][] dp = new int[n][total + 1];

    // populate the total=0 columns, as we will always have an empty set for zero toal

    for(int i=0; i < n; i++)

      dp[i][0] = 1;

    // process all sub-arrays for all capacities

    for(int i=0; i < n; i++) {

      for(int t=1; t <= total; t++) {

        if(i > 0)

          dp[i][t] = dp[i-1][t];

        if(t >= denominations[i])

          dp[i][t] += dp[i][t-denominations[i]];

      }

    }

    // total combinations will be at the bottom-right corner.

    return dp[n-1][total];

  }

  public static void main(String[] args) {

    CoinChange cc = new CoinChange();

    int[] denominations = {1, 2, 3};

    System.out.println(cc.countChange(denominations, 5));

  }

}