**Introduction**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/NE0yNJ1rZy6#introduction)

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

**Example 1:**

Denominations: {1,2,3}  
Total amount: 5  
Output: 2  
Explanation: We need minimum of two coins {2,3} to make a total of '5'

**Example 2:**

Denominations: {1,2,3}  
Total amount: 11  
Output: 4  
Explanation: We need minimum four coins {2,3,3,3} to make a total of '11'

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/NE0yNJ1rZy6#problem-statement)

Given a number array to represent different coin denominations and a total amount ‘T’, we need to find the minimum number of coins needed to make change for ‘T’. 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*).

### Top-down Dynamic Programming with Memoization [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/NE0yNJ1rZy6#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];

    int result = this.countChangeRecursive(dp, denominations, total, 0);

    return (result == Integer.MAX\_VALUE ? -1 : result);

  }

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

    // base check

    if (total == 0)

      return 0;

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

      return Integer.MAX\_VALUE;

    // check if we have not already processed a similar sub-problem

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

      // recursive call after selecting the coin at the currentIndex

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

      int count1 = Integer.MAX\_VALUE;

      if( denominations[currentIndex] <= total ) {

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

        if(res != Integer.MAX\_VALUE){

          count1 = res + 1;

        }

      }

      // recursive call after excluding the coin at the currentIndex

      int count2 = countChangeRecursive(dp, denominations, total, currentIndex + 1);

      dp[currentIndex][total] = Math.min(count1, count2);

    }

    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));

    System.out.println(cc.countChange(denominations, 11));

    System.out.println(cc.countChange(denominations, 7));

    denominations = new int[]{3, 5};

    System.out.println(cc.countChange(denominations, 7));

  }

}

### Bottom-up Dynamic Programming [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/NE0yNJ1rZy6#bottom-up-dynamic-programming)

Let’s try to populate our array dp[TotalDenominations][Total+1] for every possible total with a minimum number of coins needed.

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

* Exclude the coin: In this case, we will take the minimum coin count from the previous set => dp[index-1][t]
* Include the coin if its value is not more than ‘t’: In this case, we will take the minimum count needed to get the remaining total, plus include ‘1’ for the current coin => dp[index][t-denominations[index]] + 1

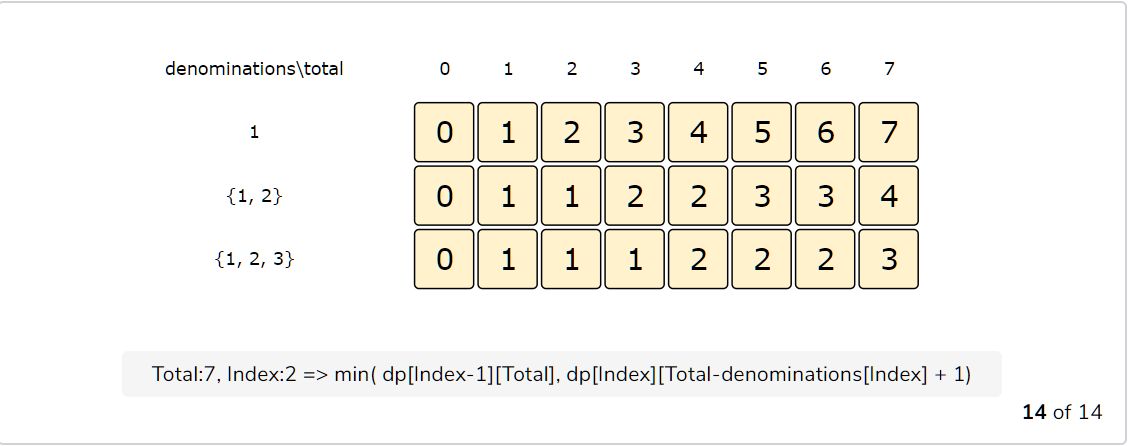
Finally, we will take the minimum of the above two values for our solution:

    dp[index][t] = min(dp[index-1][t], dp[index][t-denominations[index]] + 1)

Let’s draw this visually with the following example:

    Denominations: [1, 2, 3]    
    Total: 7

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



import java.util.Arrays;

class CoinChange {

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

  {

    int n = denominations.length;

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

    for(int i=0; i < n; i++)

      for(int j=0; j <= total; j++)

        dp[i][j] = Integer.MAX\_VALUE;

    // populate the total=0 columns, as we don't need any coin to make zero total

    for(int i=0; i < n; i++)

      dp[i][0] = 0;

    for(int i=0; i < n; i++) {

      for(int t=1; t <= total; t++) {

        if(i > 0)

          dp[i][t] = dp[i-1][t]; //exclude the coin

        if(t >= denominations[i]) {

          if(dp[i][t-denominations[i]] != Integer.MAX\_VALUE)

            dp[i][t] = Math.min(dp[i][t], dp[i][t-denominations[i]]+1); // include the coin

        }

      }

    }

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

    return (dp[n-1][total] == Integer.MAX\_VALUE ? -1 : 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));

    System.out.println(cc.countChange(denominations, 11));

    System.out.println(cc.countChange(denominations, 7));

    denominations = new int[]{3, 5};

    System.out.println(cc.countChange(denominations, 7));

  }

}