**Introduction**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQVZJx1k0WY#introduction)

We are given a ribbon of length ‘n’ and a set of possible ribbon lengths. Now we need to cut the ribbon into the maximum number of pieces that comply with the above-mentioned possible lengths. Write a method that will return the count of pieces.

**Example 1:**

n: 5  
Ribbon Lengths: {2,3,5}  
Output: 2  
Explanation: Ribbon pieces will be {2,3}.

**Example 2:**

n: 7  
Ribbon Lengths: {2,3}  
Output: 3  
Explanation: Ribbon pieces will be {2,2,3}.

**Example 3:**

n: 13  
Ribbon Lengths: {3,5,7}  
Output: 3  
Explanation: Ribbon pieces will be {3,3,7}.

### Problem Statement [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQVZJx1k0WY#problem-statement)

Given a number array to represent possible ribbon lengths and a total ribbon length ‘n’, we need to find the maximum number of pieces that the ribbon can be cut into.

This problem follows the [Unbounded Knapsack](https://www.educative.io/collection/page/5668639101419520/5633779737559040/5745865499082752/) pattern and is quite similar to [Minimum Coin Change](https://www.educative.io/collection/page/5668639101419520/5633779737559040/5668753656250368) (MCC). The only difference is that in MCC we were asked to find the minimum number of coin changes, whereas in this problem we need to find the maximum number of pieces.

class CutRibbon {

  public int countRibbonPieces(int[] ribbonLengths, int total) {

    int maxPieces = this.countRibbonPiecesRecursive(ribbonLengths, total, 0);

    return maxPieces == Integer.MIN\_VALUE ? -1 : maxPieces;

  }

  private int countRibbonPiecesRecursive(int[] ribbonLengths, int total, int currentIndex) {

    // base check

    if (total == 0)

      return 0;

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

      return Integer.MIN\_VALUE;

    // recursive call after selecting the ribbon length at the currentIndex

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

    int c1 = Integer.MIN\_VALUE;

    if( ribbonLengths[currentIndex] <= total ) {

      int result = countRibbonPiecesRecursive(

                    ribbonLengths, total - ribbonLengths[currentIndex], currentIndex);

      if(result != Integer.MIN\_VALUE){

        c1 = result + 1;

      }

    }

    // recursive call after excluding the ribbon length at the currentIndex

    int c2 = countRibbonPiecesRecursive(ribbonLengths, total, currentIndex + 1);

    return Math.max(c1, c2);

  }

  public static void main(String[] args) {

    CutRibbon cr = new CutRibbon();

    int[] ribbonLengths = {2,3,5};

    System.out.println(cr.countRibbonPieces(ribbonLengths, 5));

    ribbonLengths = new int[]{2,3};

    System.out.println(cr.countRibbonPieces(ribbonLengths, 7));

    ribbonLengths = new int[]{3,5,7};

    System.out.println(cr.countRibbonPieces(ribbonLengths, 13));

    ribbonLengths = new int[]{3,5};

    System.out.println(cr.countRibbonPieces(ribbonLengths, 7));

  }

}

The time complexity of the above algorithm is exponential O(2^{L+T})*O*(2​*L*+*T*​​), where ‘L’ represents total ribbon lengths and ‘N’ is the total length that we want to cut. The space complexity will be O(L+T)*O*(*L*+*T*).

Since this problem is quite similar to [Minimum Coin Change](https://www.educative.io/collection/page/5668639101419520/5633779737559040/5668753656250368), let’s jump on to the bottom-up dynamic programming solution.

### Bottom-up Dynamic Programming [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQVZJx1k0WY#bottom-up-dynamic-programming)

Let’s try to populate our array dp[ribbonLength][total+1] for every possible ribbon length with a maximum number of pieces.

So for every possible length ‘len’ (0<= len <= total) and for every possible ribbon length index (0 <= index < ribbonLengths.length), we have two options:

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

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

  dp[index][t] = max(dp[index-1][len], 1 + dp[index][len-ribbonLengths[index]])

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/YQVZJx1k0WY#code-2)

Here is the code for our bottom-up dynamic programming approach:

import java.util.Arrays;

class CutRibbon {

  public int countRibbonPieces(int[] ribbonLengths, int total)

  {

    int n = ribbonLengths.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.MIN\_VALUE;

    // populate the total=0 columns, as we don't need any ribbon 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) //exclude the ribbon

          dp[i][t] = dp[i-1][t];

        // include the ribbon and check if the remaining length can be cut into available lengths

        if(t >= ribbonLengths[i] && dp[i][t-ribbonLengths[i]] != Integer.MIN\_VALUE)

          dp[i][t] = Math.max(dp[i][t], dp[i][t-ribbonLengths[i]]+1);

      }

    }

    // total combinations will be at the bottom-right corner, return '-1' if cutting is not possible

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

  }

  public static void main(String[] args) {

    CutRibbon cr = new CutRibbon();

    int[] ribbonLengths = {2,3,5};

    System.out.println(cr.countRibbonPieces(ribbonLengths, 5));

    ribbonLengths = new int[]{2,3};

    System.out.println(cr.countRibbonPieces(ribbonLengths, 7));

    ribbonLengths = new int[]{5,3,7};

    System.out.println(cr.countRibbonPieces(ribbonLengths, 13));

    ribbonLengths = new int[]{3,5};

    System.out.println(cr.countRibbonPieces(ribbonLengths, 7));

  }

}

[[1]](#footnote-1)

The above solution has time and space complexity of O(L\*N)*O*(*L*∗*N*), where ‘L’ represents total ribbon lengths and ‘N’ is the total length that we want to cut.

1. [↑](#footnote-ref-1)