Complexity discussion:

About the problem:

Consider a box with three balls, each with a different value: ball A is worth 2 points, ball B is worth 3 points and ball C is worth 4 points. Write a function that calculates the number of different ways of reaching a total sum of N points by extracting from the box one ball at a time, taking into account the order in which the balls are drawn.

For example, for N=6 you have 4 different possibilities:

- Taking ball B twice in a row → 3 + 3 = 6 points
- Taking ball A three times in a row → 2 + 2 + 2 = 6 points
- Taking ball A, then ball C → 2 + 4 = 6 points
- Taking ball C, then ball A → 4 + 2 = 6 points

Approach

First of all we need to identify the problem at hand. What we are asked to solve is a variation of the *Coin change problem*, which consists of determining all the possible ways to return a certain amount of change with three different-valued kinds of coins (assuming an infinite supply of coins). On our variation we are dealing with balls in a box instead of coins and we need to reach an arbitrary number that takes the place of the change. However, most importantly, in this variation we want to consider all the possible combinations from the same set of balls as different, for example, will count as two different ways the combinations (a + b) and (b+a).

Regarding the implementation of the solution, two different approaches have been coded for the sake of comparison and explanation; the recursive approach and the dynamic programming approach.

The recursive approach seems to be the most intuitive. The idea is that, starting from zero as the root, on each node will have three sons, one for each possible ball that you can pick or value that you can sum. The nodes will contain the value from the father plus the value of the ball that has been chosen. This process will be repeated until we reach the leaves, that will be either adding to the desired number N or adding to more. In order to return the number of combinations that add to N we now just need to count the number of leaves that reach exactly N.

The dynamic programming approach consists of building on top of the previous N number of possibilities to define the desired (current) N. For example, if we know that there are 4 ways to reach N=6 and that there is a ball that can add 2, then we know that we can reach N=8 with these 4 ways too by just adding 2. Then, we need to consider all the other possible ways to reach N=8, which in our case would be coming from N=5 and adding 3 and from N=4 and adding 4. Alternatively we can annotate this as f(i) = f(i-2)+f(i-3)+f(i-4), where the result we are trying to compute is f(N). In order to implement this we define an array that represents f(i) and add a left padding of zeros so that we can access the negative values of the function when computing f(1 to 3). Finally we define f(0) as 1 as the starting point. Finally we iterate using the function previously stated and retrieve the last value of the array.

Complexity

For the recursive approach, each node is branched into 3 until reaching N or more, which leads to an exponential computational complexity of O(k^n). On the bright side, no additional space is required for this computation; space complexity is O(1).

On the other hand, the dynamic programming approach requires a space of N+4, spatial complexity of O(n), but can reach the result on a single iteration of an array, which greatly improves the computational complexity from exponential to linear: O(n). Therefore, since we are trying to optimize the computational complexity of the solution I would suggest using the dynamic programming approach.