Dear reader, welcome to the article on the problem named **‘**[**Coin Change - Permutations**](https://www.pepcoding.com/resources/online-java-foundation/dynamic-programming-and-greedy/coin-change-permutations-official/ojquestion)**’.**

It is a slight variation of ‘[*Target Sum Subsets*](https://www.pepcoding.com/resources/online-java-foundation/dynamic-programming-and-greedy/target-sum-subsets-dp-official/ojquestion)’, and just a line of change in code from ‘[*Coin Change - Combinations*](https://www.pepcoding.com/resources/online-java-foundation/dynamic-programming-and-greedy/coin-change-combination-official/ojquestion)’. I recommend you to solve these 2 previous problems first if you have landed on this problem directly.

***Problem Statement:***

You are given an array of n numbers, which represents n different types of ***denominations of coins***. You are given a target ‘*amt*’, you need to count the number of permutations of the coins possible that sum up to the given target amount, i.e. in how many ways you can pay ‘*amt’* using the available denominations.

***Note 1***: You can consider that you are having ***infinite supply*** of each coin denomination i.e. the same coin denomination can be used for many installments in payment of "amt".

***Note 2***: You are required to find the count of ***distinct*** ***permutations***. For eg, 2 + 2 + 3 = 7 and 2 + 3 + 2 = 7 and 3 + 2 + 2 = 7 are different permutations, hence their count will be 3. However, we cannot swap coin value = 2 with other coin value = 2 only to form another permutation, as they will not form distinct permutations.

*Example*, to pay the amount = 7 using coins {2, 3, 5, 6}, there are five coin permutations possible: (2, 5), (5, 2), (2, 2, 3), (2, 3, 2) and (3, 2, 2). Hence the answer is 5.

*Note*: If you have not tried enough to come up with logic, then we recommend you to first spend an hour or so doing it, else read only the logic used, take it as a hint and try the problem again with the same logic.

***Solution:***

So, as I said, this problem is a slight variation of the ‘*Target Sum Subsets*’ problem. Can you figure out the similarities and differences between the two questions?

***Similarity***: We have to find a subset of the array (coins in this case) whose sum is equal to the target (amount in this case).

***Difference***: In the target sum subset, we can take each element at most one time. But in this problem, we can take each coin any number of times (infinite supply).

* Let us define our ***dp state on a 1d array***. We will create an array ***of size (amount + 1)*** all ***filled with 0 initially***.
* Value at index i in the dp array represents the count of ways to have permutations of coins such that their sum of values is i.

***Trivial Case***: What is the number of ways to select coins so that their sum = 0. Interestingly, there is ***1 way to have sum = 0***, and that way is to ***select no coins***. Hence, we will ***initialize dp[0] as 1***.

Until now, whatever we have done is identical to what we did in coin change combinations.

Q) But how did we stop coin permutations in the previous solution?

* We ran the loop for the first coin for the entire array at once and then ran the loop for the second coin for the entire array, and so on.
* Summary is that in the previous solution, outer loop i went from 0 to n and inner loop went from 1 to amount.
* By doing so, we stopped the occurrence of the first coin after the second coin.

Q) How can we bring back those permutations? Try to modify the code a little bit, so that first all denominations of coins are analyzed for smaller amounts, then for larger amounts.

* So, in this case, what we all need to do is first find ways of permuting coins for amount 1 using all coins, then for amount 2, and so on.
* Hence we need the amount loop to be the outer one and the loop over coins to become the inner one.
* *Voila*! Only thing we need to change is **interchange the loops**. That’s All.

***Algorithm***

* We will run outer loop i from 1 to amount + 1, i.e. through the ***dp array***.
* Now for amount i, we will run a ***nested loop*** through all coin denominations, i.e. the inner loop j will iterate from 0 to n-1.
* If we pick the coin arr[j], then the remaining sum becomes i - arr[j]. Hence, the number of ways to form sum = i with the last coin as arr[j] is the same as the number of ways to form sum = i - arr[j].
* Thus, we will add the number of ways of combination of coins to form sum = i - arr[j] to the current dp state, i.e. dp[i].

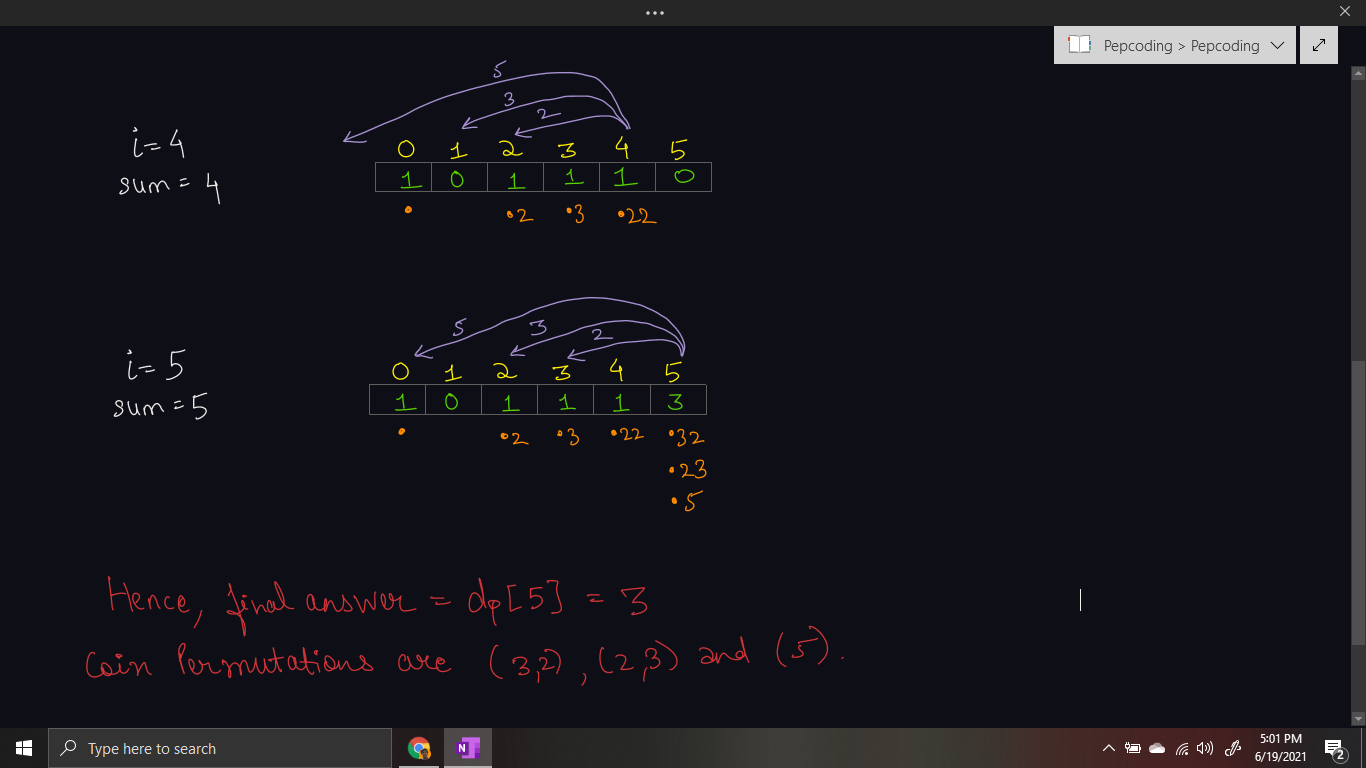
**dp[i] += dp[i - arr[j]] (if arr[j] <= i)**

* At last, when all coins are considered and the entire dp array is filled, we will ***return the value of dp[amt]*** as it contains the number of ways of coin change permutations.

***Note***:

* Before accessing the value of i - arr[j], we will ***check if arr[j] >= i*** or not.
* This is because, if we take the current coin arr[j] which is greater than the sum i required, then the remaining sum will become negative.
* We do not have coins with negative denominations, and also checking dp[i - arr[j]] when i < arr[j] will give a run-time error (due to out of bound index).

Let us try to fill the dp table using the algorithm stated above:



***Implementation***

*Note*: Before reading the Code, we recommend that you must try to come up with the solution on your own. Now, hoping that you have tried by yourself, here is the Java code.

import java.io.\*;

import java.util.\*;

public class Main {

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n = Integer.parseInt(br.readLine());

int[] coins = new int[n];

for (int i = 0; i < n; i++) {

coins[i] = Integer.parseInt(br.readLine());

}

int amt = Integer.parseInt(br.readLine());

int[] dp = new int[amt + 1];

dp[0] = 1;

for (int i = 1; i < dp.length; i++) {

for (int coin : coins) {

if (i >= coin) {

dp[i] += dp[i - coin];

}

}

}

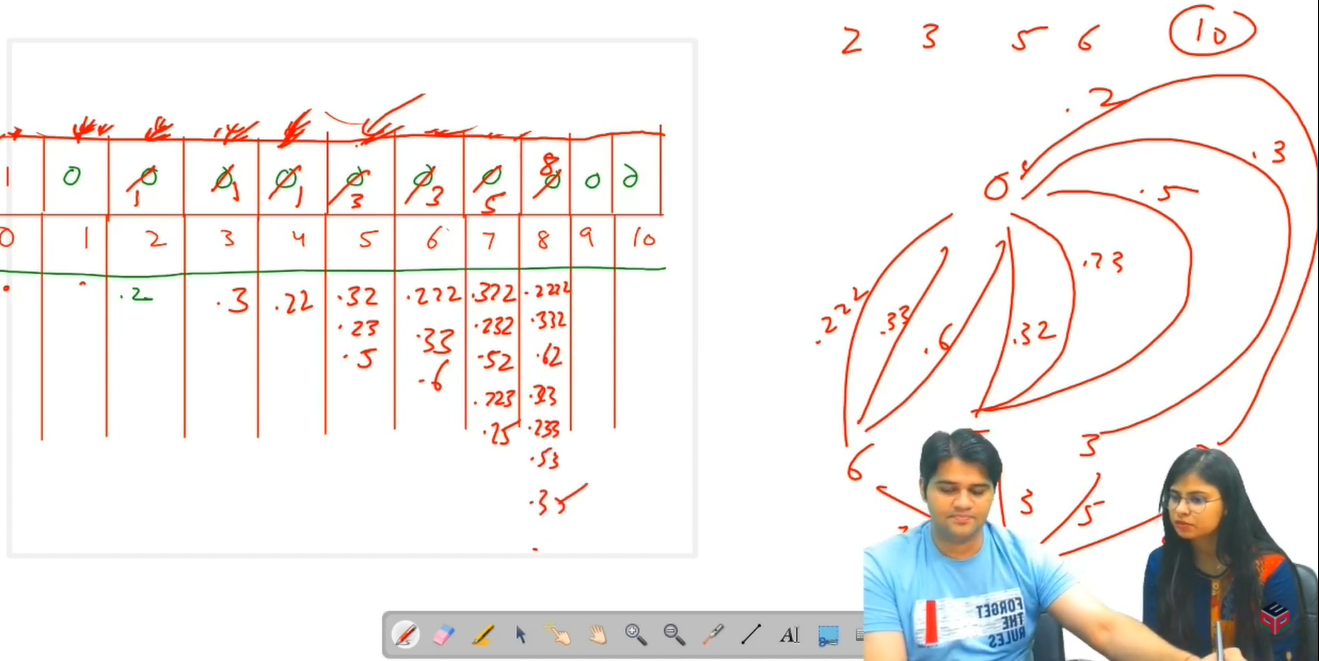
System.out.println(dp[amt]);

}

}

Java Code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=yc0LunmJA1A) from timestamp *[18:05, 23:45]*.

**Dry run** for a sample test case is shown in the same video. Please watch the complete video to analyze it completely.



* What is the ***time complexity*** of the above code?

Since we are running two nested loops, outer loop from 1 to amount, and the inner loop from 0 to n-1 (through coins denominations), hence the time complexity will be ***O(n \* amount)***.

* What is the ***space complexity*** of the above code?

We used a 1D dp array of size = amount + 1 to count all the permutations possible. Hence, auxiliary space of ***O(amount)*** is used.

***Follow Up:***

Q) We have counted all the permutations in this problem. How can we print all those permutations ? How will the space complexity of the solution change?

R) Yes, it is very simple to print all the ways also. We can create an array of strings instead of 1d arrays and at each index store all the strings (coin permutations) possible instead of just the count of such permutations. But be sure to add an empty string ( represented by . in the dry run), so that algorithm can generate all possible permutations.

***Asked in Companies***: *Uber*

Now, please watch a [***comparison video***](https://www.youtube.com/watch?v=JyyS9mlMcr4) between the 3 problems discussed so far: Target Sum Subset, Coin Change - Combinations and Coin Change - Combinations.

Next problems are based on ***Knapsack*** and you will be glad to know that they are also a slight variation of ‘***Target Sum Subset***’.

Subscribe to Pepcoding’s youtube channel for more such amazing content on Data Structures & Algorithms and follow the resources available for all students in the resources section of Pepcoding’s website.

You can suggest any improvements to the article on our telegram channel, or on the youtube channel’s comment section.

Article Contributed by:

Archit Aggarwal