Dear reader, welcome to the article on the problem named **‘**[**Coin Change - Combinations**](https://www.pepcoding.com/resources/online-java-foundation/dynamic-programming-and-greedy/coin-change-combination-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)’. I recommend you to solve the previous problem 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 combinations 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 combinations and ***not permutations***. For eg, 2 + 2 + 3 = 7 and 2 + 3 + 2 = 7 and 3 + 2 + 2 = 7 are different permutations of the same combination. You should treat them as 1 and not 3.

You can also watch the [question video](https://www.youtube.com/watch?v=Ph1EB07Q4pA) to understand the problem statement in more detail.

*Example*, to pay the amount = 7 using coins {2, 3, 5, 6}, there are two combinations of coins possible: {2, 5} and {2, 2, 3}. Hence answer is 2.

*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 combinations 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***.

***Algorithm***

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

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

* 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 combinations.

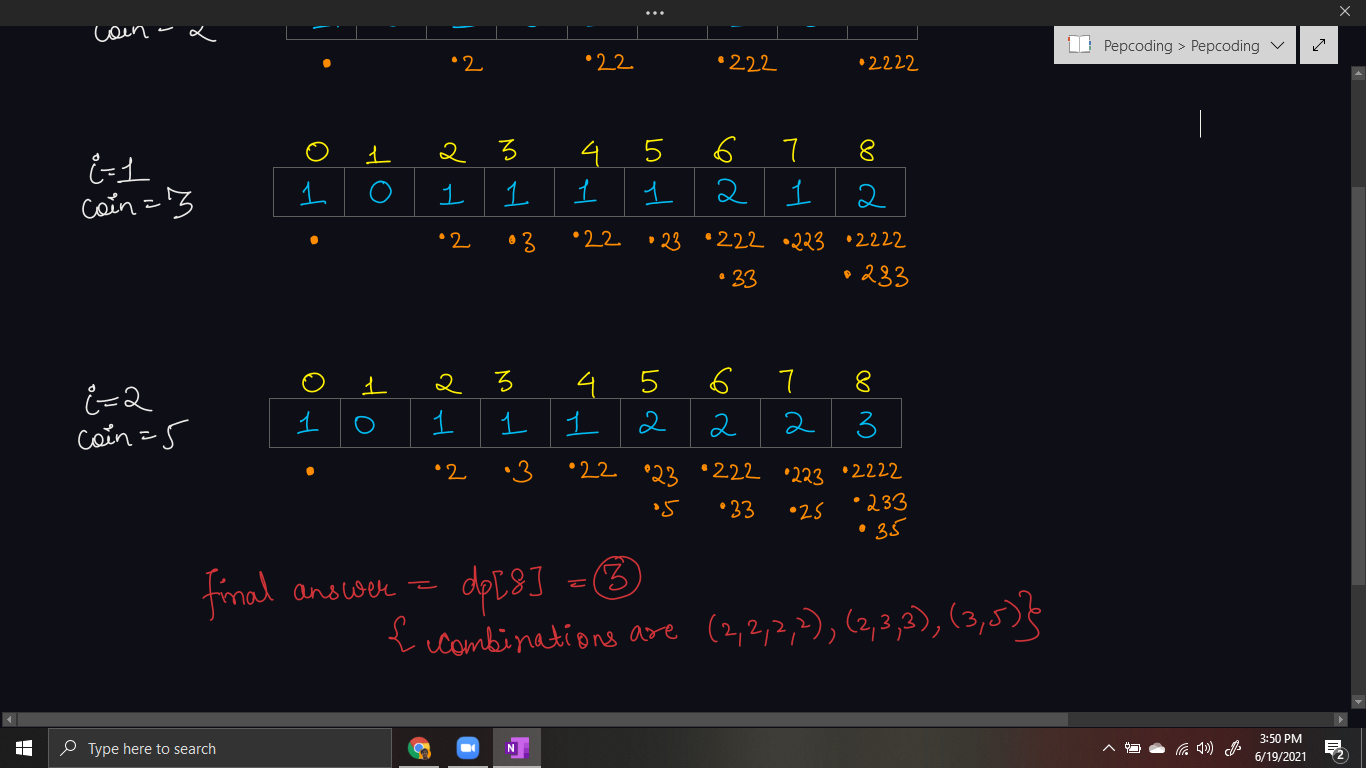
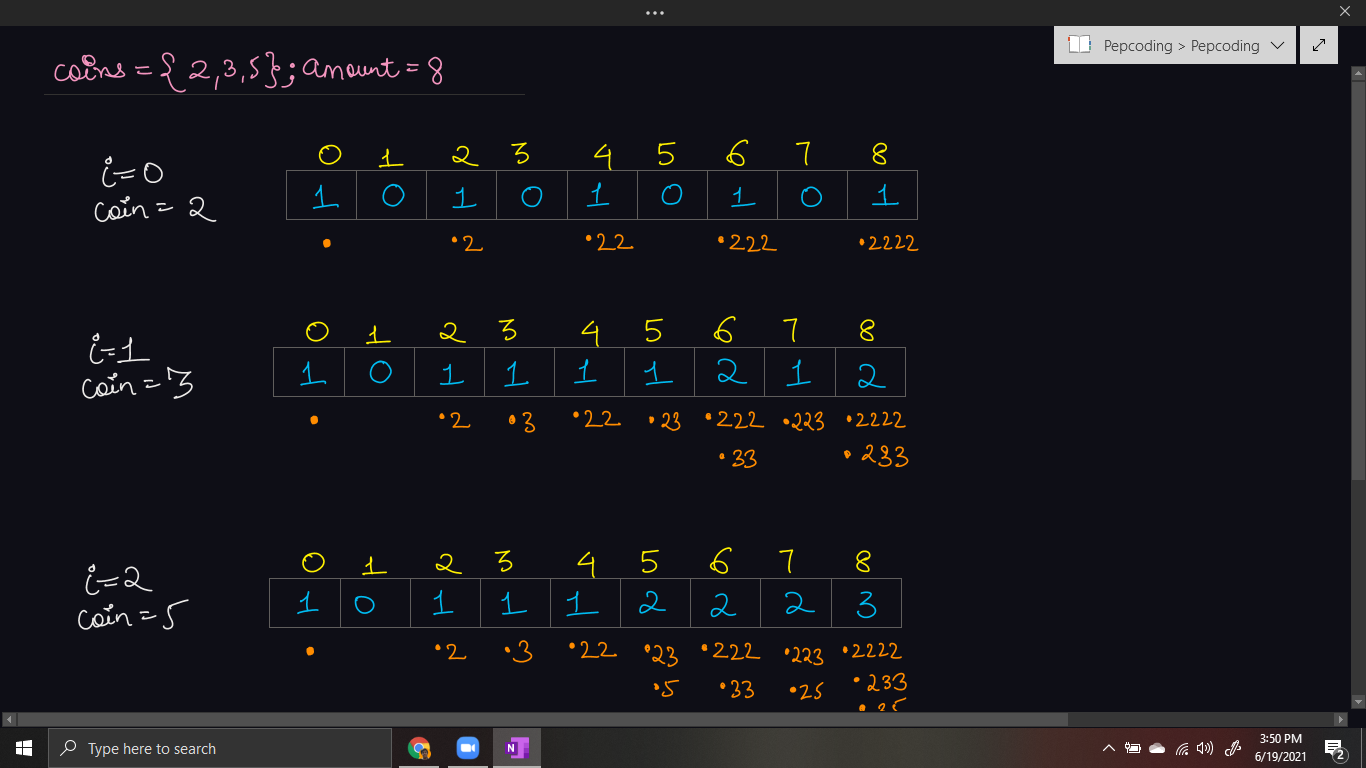
***Note***:

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

**Q) How are we restricting permutations?**

* Please look carefully, that we run the outer loop for coin denominations, and then the inner loop through the dp array.
* Hence, first all ways will get added to the dp array (for all amounts = 1 to amt), using the current coin. Then, in the second iteration, all ways will get added to the dp array using the second coin, and so on.
* Since, the first coin will not come again after the second coin for a given dp state, we are successfully able to prevent permutations of coins.

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 coin: coins){

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

if(i >= coin){

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

}

}

}

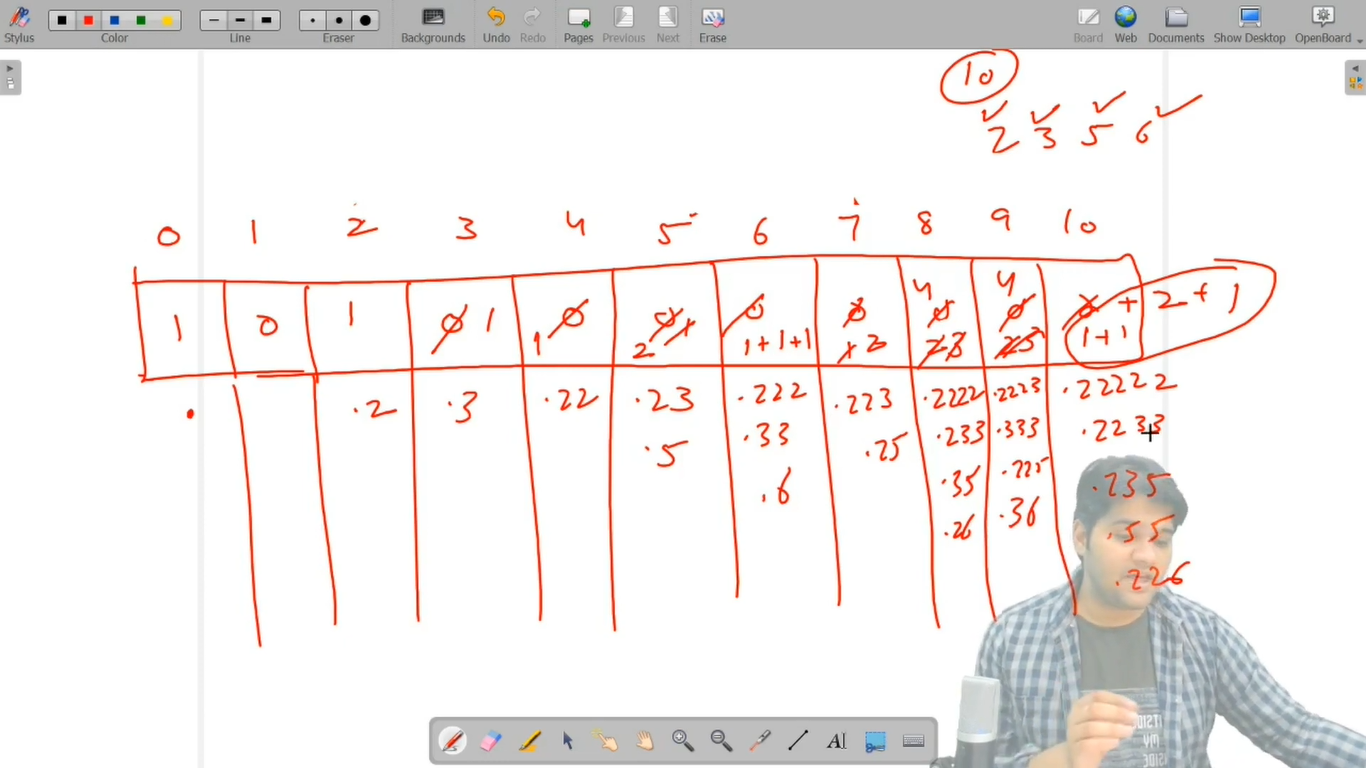
System.out.println(dp[amt]);

}

}

Java Code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=l_nR5X9VmaI) from timestamp *[9:45, 16:05]*.

**Dry run** for a sample test case is shown in the same video from timestamp *[16:05, 21:50]*.



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

Since we are running two nested loops, one on the types of denominations, i.e. n, and the inner loop from 1 to amount, 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 combinations possible. Hence, auxiliary space of ***O(amount)*** is used.

***Follow Up:***

Q) We have counted all the combinations in this problem. How can we print all those combinations? 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 combinations) possible instead of just the count of such combinations. But be sure to add an empty string ( represented by . in the dry run), so that algorithm can generate all possible combinations.

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

Next problem: ‘***Coin Change - Permutations***’ is very similar to this problem. Just you need to figure out the way so that we can find out the permutations of a given combination as well.

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