Dear reader, welcome to the article on the problem named **‘Target Sum Subset - DP’.** This problem is very interesting, and there are many variations to this problem which we will see in continuation and try to solve using the concept we learn here.

***Problem Statement:***

* Given an array of n integers (non-negative), and a target value *tar,* you need to *check whether a subset of the array whose sum of elements is equal to the target tar exists or not*.
* Please recall that a subset (or subsequence) of an array is taking zero or more elements from the array (in the same order in which they occur in the array).
* Here, you need to just print *true or false* whether such a target sum subset exists or not. You do not need to print the subset.

You can also watch the [question video](https://www.youtube.com/watch?v=CqEsqfidKi0) to understand the problem statement in more detail.

*Example*, for array = {4, 2, 7, 1, 3} and target = 10, the answer will be *true* as there exists a subset {7, 3} whose sum = target.

There can be other subsets also like {4, 2, 1, 3} whose sum = target = 10, but as soon as we find any one subset, we can print true.

*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 1: Using Recursion & Backtracking***

* Have you seen a similar question earlier? Yes, we solved the problem to print all subsets whose sum becomes equal to target using [*recursion & backtracking*](https://www.pepcoding.com/resources/online-java-foundation/recursion-backtracking/target-sum-subsets-official/ojquestion).
* We can modify the solution to just print true or false if we get at least one such subset, otherwise false.
* It’s time complexity will be, as discussed, exponential in nature, i.e. ***O(2^n)***, as we have *two* options for each element, whether to include it in the subset so far or not include it.

Can we do something better than an exponential time backtracking solution?

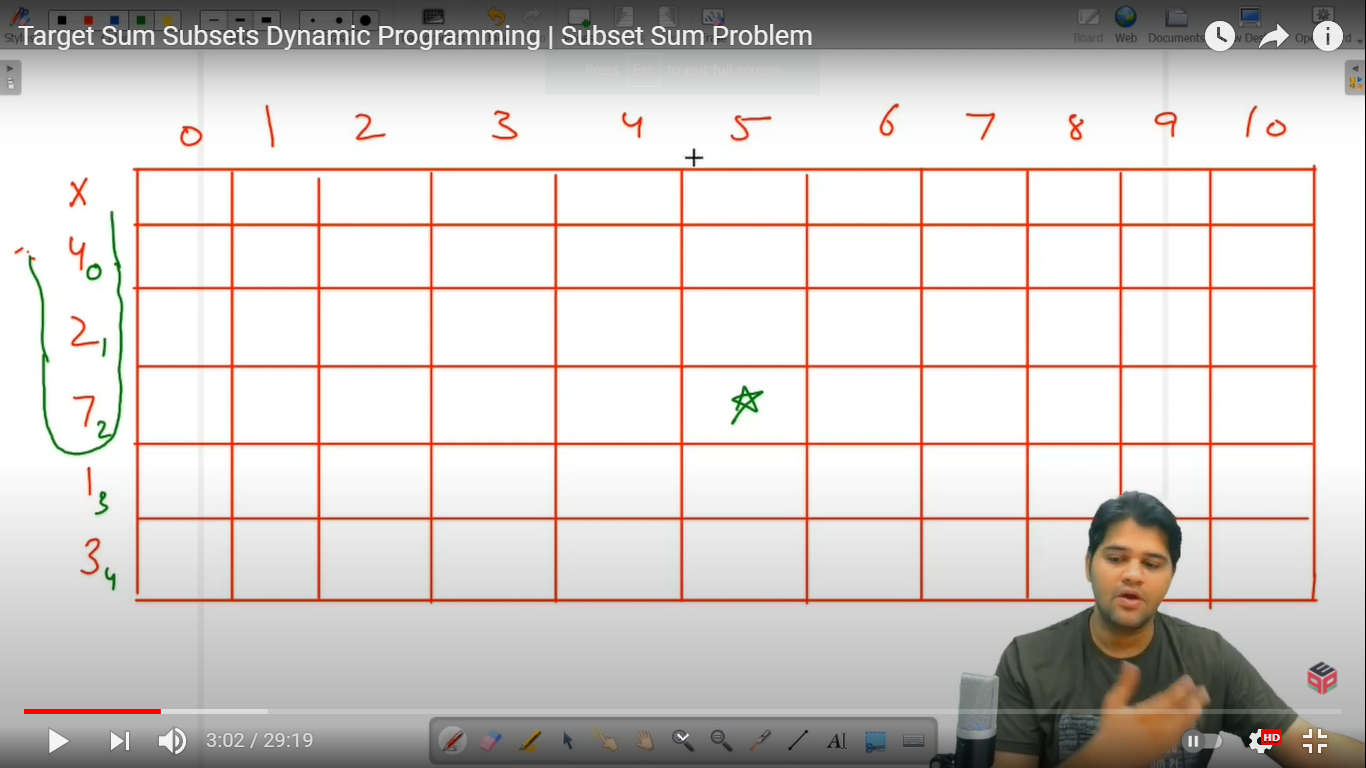
***Solution 2: Using Dynamic Programming***

Let us first discuss ***HOW*** we will solve this problem using DP. After understanding the algorithm, we will analyze ***WHY*** we did so.

We will make a ***2D array*** *(dp tabulation array)*of size ***((n + 1)* X *(tar + 1))*** where n = size of array and tar is the target value. Hence n + 1 will be the number of *rows*, whereas tar + 1 will be the number of *columns*.

What is the meaning of cell ***dp[i][j]***? It represents whether we can make a subset from array elements of index {0, 1, … i-1} whose sum is equal to j.

For Example)



The cell marked with \*, represents whether there exists a subset of {4, 2, 7} (which is {arr[0], arr[1], arr[2]}) whose sum is equal to 5.

* The ***first row*** represents an ***empty array***. The first cell in the first row (dp[0][0]) represents whether the empty array has sum = 0 or not. Second cell in the first row (dp[0][1]) represents whether the empty array has sum = 1 or not, and so on. Since the empty array has sum = 0, only the first cell will be true, rest all cells in the first row will be false.
* The **first column** represents whether there is a subset with **target** **= 0**. Since every prefix array (array from index 0 to i) can have an empty subset whose target = 0, hence all cells in the first column can be marked as *true*.
* Now, we will run a loop for every array element (representing the rows in dp table). We will run another nested loop from target value 1 to sum.
* Hence, at row = i and column = j, we are checking whether there exists a subset from the prefix array {0, 1, … i-1} whose sum = j.
* We can have two options for the current element: whether to include it in the subset formed by the elements before it, or not to include it in the subset.
* If we include the array element, then we have to check if there exists a subset from previous array elements, whose sum = j - arr[i-1]. Hence, we will check if ***j - arr[i-1] >= 0*** (to avoid out of bound index) and if ***dp[i-1][j - arr[i-1]] = true***, then we make dp[i][j] as true.
* If we do not include the array element, then we will have to check if there exists a subset from previous array elements whose sum = j itself. Hence, we will check if ***dp[i-1][j] = true***, then we will make dp[i][j] as true.
* In the last, we will check if the whole array can make a subset whose sum = target or not. That is, we return the value of ***dp[i][tar]***.

***Important Note***:

**DP[i][j] = DP[i-1][j] OR DP[i-1][j - arr[i-1]]**

1. Please remember that the first row in the dp table represents zero elements or empty subarray, hence row ***dp[i] corresponds to arr[i-1]*** element.
2. Also remember, for checking the value of *dp[i-1][j - arr[i-1]]*, we are first checking ***whether j - arr[i-1] >= 0*** or not. If arr[i-1] is greater than j, then the remaining sum will become negative, but we do not have any index representing a negative target.

***Analogy to Cricket***:

* You can try to think of this problem analogous to a cricket match.
* Try to think that the array elements are the runs that can be scored by a player, and make target sum analogous to the total score required to chase.
* Making a subset of an array is analogous to picking a team from the squad.
* Hence for every player, we will check if the team without him/her can score runs = j or not.
* If he is included in the team, then we will check if the rest of the team can score the rest of the runs required to achieve total or not, i.e. (j - arr[i-1]).

***WHY are we using this approach?***

If you will carefully see, then we are not re-computing all the subsets from the prefix array of previous elements, i.e. we are not generating all the subsets first and then choosing whether to add the current element to them or not. Instead, we are just using the result of whether any subset of previous elements can have a given sum.

Hence, we are reducing the exponential time complexity to somewhat ***pseudo polynomial time complexity***.

*Note*: We are saying it as pseudo polynomial time taking, as the time complexity will also depend on the value of target, which in turn does not depend on the number of elements (n).

***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[] arr = new int[n];

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

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

}

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

boolean[][] dp = new boolean[arr.length + 1][tar + 1];

for (int i = 0; i < dp.length; i++) {

for (int j = 0; j < dp[0].length; j++) {

if (i == 0 && j == 0) {

dp[i][j] = true;

} else if (i == 0) {

dp[i][j] = false;

} else if (j == 0) {

dp[i][j] = true;

} else {

if(dp[i - 1][j] == true){

dp[i][j] = true;

} else {

int val = arr[i - 1];

if (j >= val

&& dp[i - 1][j - val] == true) {

dp[i][j] = true;

}

}

}

}

}

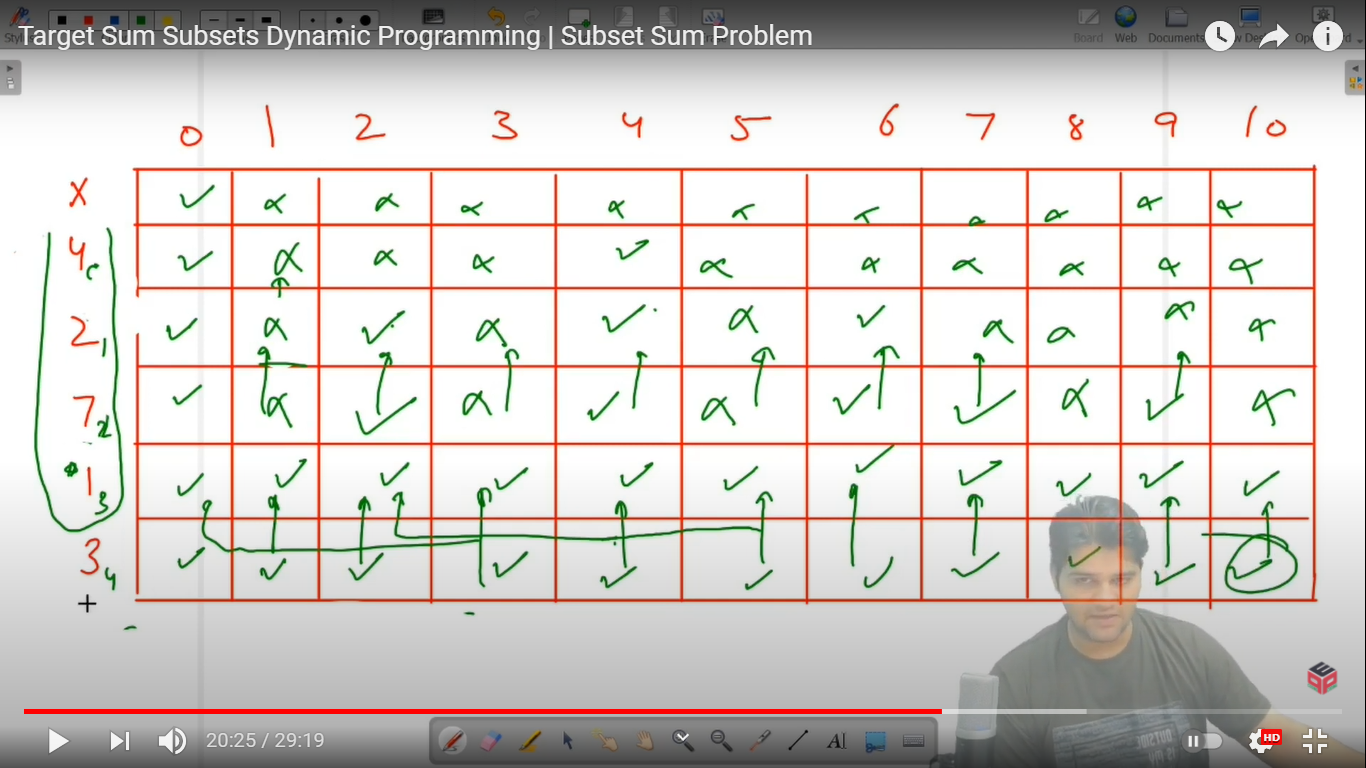
System.out.println(dp[dp.length - 1][tar]);

}

}

Java Code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=tRpkluGqINc) from timestamp *[22:10, 28:55]*.

The 2D DP table is shown for the example test case in the same video in detail. Please watch the ***complete video*** to have a better understanding.



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

Since we are running two nested loops, one from 1 to n, inner loop from 1 to target, hence the time complexity will be ***O(n \* target)***.

This problem was a *NP-Complete Problem*. The value of the target does not depend on the number of elements n.

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

Since we used a 2D DP array of size (n + 1) \* (target + 1), hence there is ***O(n \* target) auxiliary space*** required.

***Follow Up:***

1. This solution will only work if the array elements are ***non-negative,*** as we are representing column index as sum, which cannot be negative. Can you think of a solution which will work for both negative and positive integers?
2. Can you try to reduce the space complexity of the solution? How about a solution which takes only ***O(target) auxiliary space***?
3. Can you modify the algorithm to ***print*** any one such ***subset*** whose sum = target? Also, can you print all such subsets using the modified 2D DP array?

***Asked in Companies***: *Amazon, Adobe*

Hope that you liked the article on ***Target Sum Subset***. Next few problems: Coin-Change Permutation, Coin-Change Combination, 0-1 Knapsack and Unbounded Knapsack are based on the technique discussed in this question. So, please solve these problems as a group consecutively.

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