Target Sum

Problem Description:

We are given an array ‘ARR’ of size ‘N’ and a number ‘Target’. Our task is to build an expression from the given array where we can place a ‘+’ or ‘-’ sign in front of an integer. We want to place a sign in front of every integer of the array and get our required target. We need to count the number of ways in which we can achieve our required target.

Example:

Target: 3

+1 -2 +3 +1 = 3

+1 -2 +3 +1 = 3

There are **2** ways

Disclaimer: *Don’t jump directly to the solution, try it out yourself first.*

Pre-req: Count Partitions with Given Difference (DP – 18)

Solution :

The first approach that comes to our mind is to generate all subsequences and try both options of placing ‘-’ and ‘+’ signs and count the expression if it evaluates the answer.

This surely will give us the answer but can we try something familiar to the previous problems we have solved?

The answer is **yes!** We can use the concept we studied in the following article Count Partitions with Given Difference (DP – 18).

The following insights will help us to understand intuition better:

- If we think deeper, we can say that the given 'target' can be expressed as addition of two integers (say S1 and S2).

  S1 + S2 = target – (i)

- Now, from where will this S1 and S2 come? If we are given the array as **[a,b,c,d,e],** we want to place '+' or '-' signs in front of every array element and then add it. One example is :

  +a-b-c+d+e which can be written as **(+a+d+e) +** (-b-c)**.**

  Therefore, we can say that **S1=(+a+d+e)** and **S2=(-b-c)** for this example.

- If we calculate the total sum of elements of the array (say totSum), we can can say that,

  S1 = totSum – S2 – (ii)

- Now solving for equations (i) and (iii), we can say that

  S2 = (totSum – target)/2 – (iv)

  Therefore this question is modified to "Count Number of subsets with sum (totSum – target)/2 ". This is exactly what we had discussed in the article Count Subsets with Sum K.

  Edge Cases:

  The following edge cases need to be handled:

- As the array elements are positive integers including zero, we don't want to find the case when S2 is negative or we can say that totSum is lesser than D, therefore if totSum<target, we simply return 0.
- S2 can't be a fraction, as all elements are integers, therefore if totSum – target is odd, we can return 0.

From here on we will discuss the approach to “Count Subsets with Sum K” with the required modifications. Moreover, as the array elements can also contain 0, we will handle it as discussed in part-1 of this article.

Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in Dynamic Programming Introduction.

Step 1: Express the problem in terms of indexes.

The array will have an index but there is one more parameter “target”. We are given the initial problem to find whether there exists in the whole array a subsequence whose sum is equal to the target.

So, we can say that initially, we need to find(n-1, target) which means that we are counting the number of subsequences in the array from index 0 to n-1, whose sum is equal to the target. Similarly, we can generalize it for any index ind as follows:

```
f(ind,target) ->  Count Number of subsequences  that exists in the
                  Array from index 0 to ind, whose sum is equal to target
```

Base Cases:

- If target == 0, it means that we have already found the subsequence from the previous steps, so we can return 1.
- If ind==0, it means we are at the first element, so we need to return arr[ind]==target. If the element is equal to the target we return 1 else we return 0.

```
f(ind,target) {
    if(target==0)   return true
    if( ind==0) return arr[ind] == target



    }
```

Step 2: Try out all possible choices at a given index.

We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video "Recursion on Subsequences".

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index element. For this, we will make a recursive call to f(ind-1,target).
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current index as element as part of subsequence. As we have included arr[ind], the updated target which we need to find in the rest if the array will be target – arr[ind]. Therefore, we will call f(ind-1,target-arr[ind]).

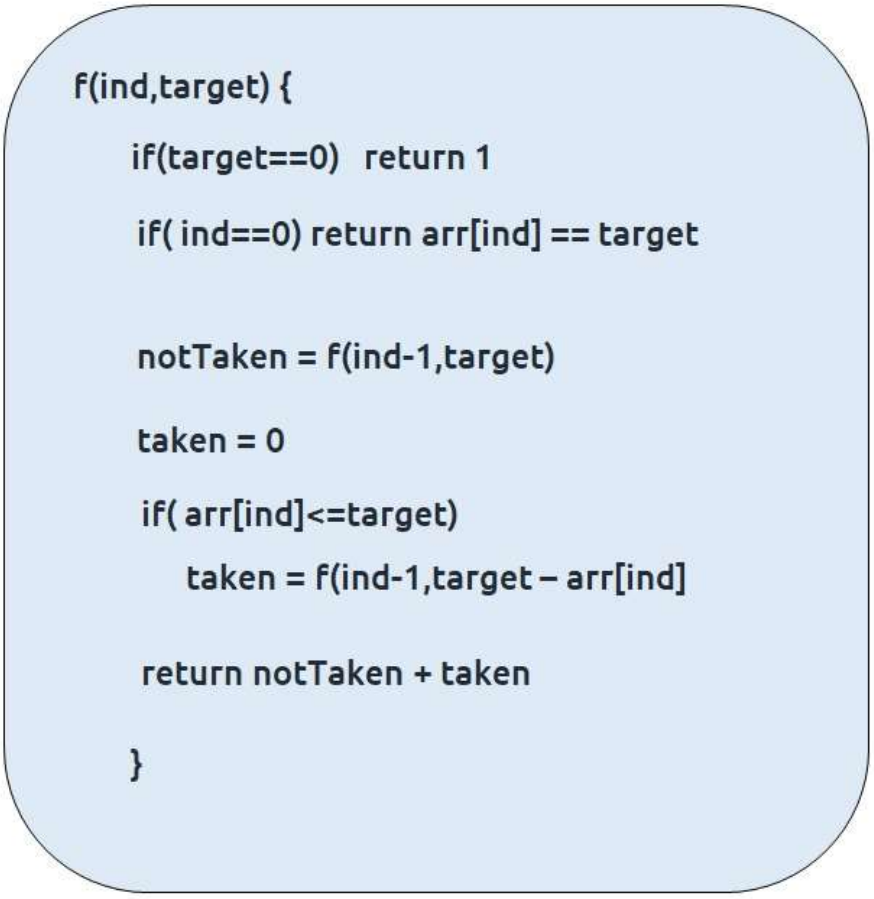
Note: We will consider the current element in the subsequence only when the current element is less than or equal to the target.

```
f(ind,target) {
    if(target==0)   return 1
    if( ind==0) return arr[ind] == target

    notTaken = f(ind-1,target)
    taken = 0
    if( arr[ind]<=target)
        taken = f(ind-1,target – arr[ind]

    }
```

Step 3: Return sum of taken and notTaken

As we have to return the total count of subsets with the target sum, we will return the sum of taken and notTaken from our recursive call.

The final pseudocode after steps 1, 2, and 3:



Steps to memoize a recursive solution:

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [n][k+1]. The size of the input array is ‘n’, so the index will always lie between ‘0’ and ‘n-1’. The target can take any value between ‘0’ and ‘k’. Therefore we take the dp array as dp[n][k+1]
2. We initialize the dp array to -1.
3. Whenever we want to find the answer of particular parameters (say f(ind,target)), we first check whether the answer is already calculated using the dp array(i.e dp[ind][target]!= -1). If yes, simply return the value from the dp array.
4. If not, then we are finding the answer for the given value for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[ind][target] to the solution we get.

Code:

- C++ Code
- Java Code

```cpp
#include <bits/stdc++.h>

using namespace std;

int countPartitionsUtil(int ind, int target, vector<int>& arr, vector<vector<int>>
&dp){

     if(ind == 0){
        if(target==0 && arr[0]==0)
            return 2;
        if(target==0 || target == arr[0])
            return 1;
        return 0;
    }

    if(dp[ind][target]!=-1)
        return dp[ind][target];

    int notTaken = countPartitionsUtil(ind-1,target,arr,dp);

    int taken = 0;
    if(arr[ind]<=target)
        taken = countPartitionsUtil(ind-1,target-arr[ind],arr,dp);

    return dp[ind][target]= (notTaken + taken);
}

int targetSum(int n,int target, vector<int>& arr){
    int totSum = 0;
    for(int i=0; i<arr.size();i++){
        totSum += arr[i];
    }

    //Checking for edge cases
    if(totSum-target<0) return 0;
    if((totSum-target)%2==1) return 0;

    int s2 = (totSum-target)/2;

    vector<vector<int>> dp(n,vector<int>(s2+1,-1));
    return countPartitionsUtil(n-1,s2,arr,dp);
}
```

```
int main() {

  vector<int> arr = {1,2,3,1};
  int target=3;

  int n = arr.size();
  cout<<"The number of ways found is " <<targetSum(n,target,arr);
}
```

Output:

The number of ways found is 2

Time Complexity: O(N*K)

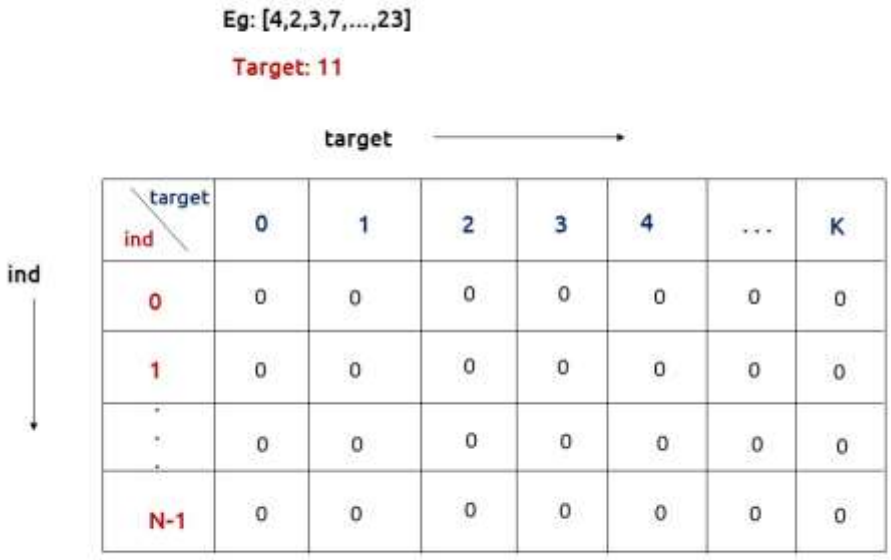
Reason: There are N*K states therefore at max ‘N*K’ new problems will be solved.

Space Complexity: O(N*K) + O(N)

Reason: We are using a recursion stack space(O(N)) and a 2D array (O(N*K)).

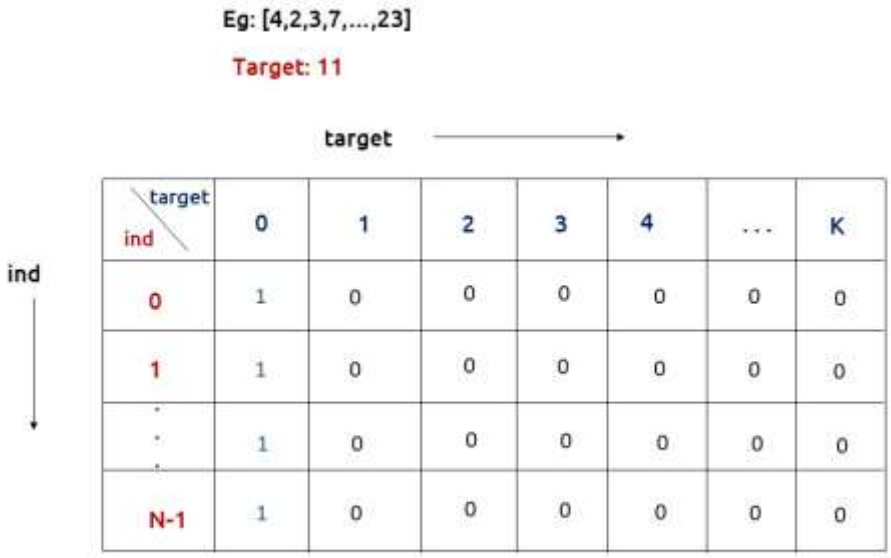
Steps to convert Recursive Solution to Tabulation one.

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

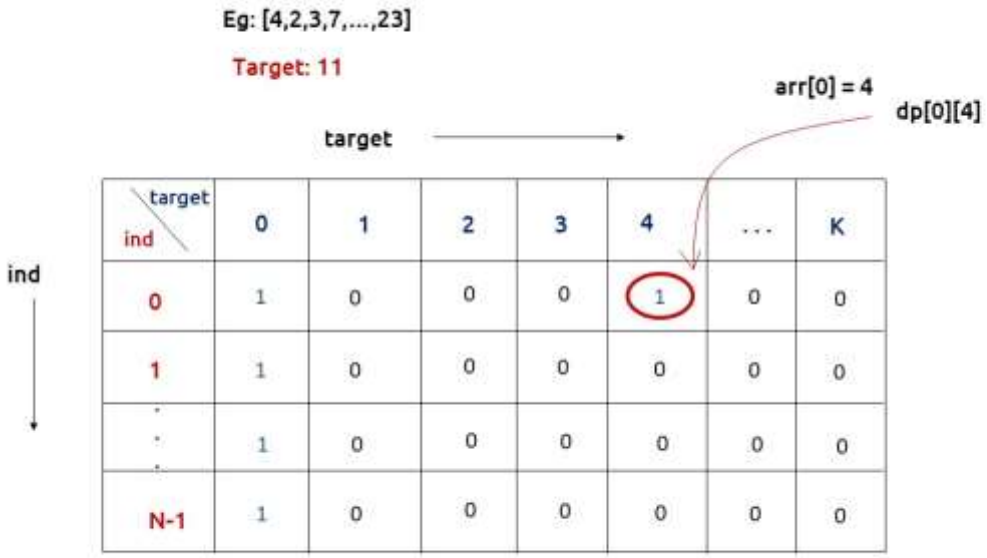


First, we need to initialize the base conditions of the recursive solution.

- If target == 0, ind can take any value from 0 to n-1, therefore we need to set the value of the first column as 1.



- The first row dp[0][] indicates that only the first element of the array is considered, therefore for the target value equal to arr[0], only cell with that target will be true, so explicitly set dp[0][arr[0]] =1, (dp[0][arr[0]] means that we are considering the first element of the array with the target equal to the first element itself). Please note that it can happen that arr[0]>target, so we first check it: if(arr[0]<=target) then set dp[0][arr[0]] = 1.



- After that , we will set our nested for loops to traverse the dp array and following the logic discussed in the recursive approach, we will set the value of each cell. Instead of recursive calls, we will use the dp array itself.
- At last we will return dp[n-1][k] as our answer.

Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

int mod =(int)1e9+7;

int findWays(vector<int> &num, int tar){
     int n = num.size();

    vector<vector<int>> dp(n,vector<int>(tar+1,0));

    if(num[0] == 0) dp[0][0] =2;  // 2 cases -pick and not pick
    else dp[0][0] = 1;  // 1 case - not pick

    if(num[0]!=0 && num[0]<=tar) dp[0][num[0]] = 1;  // 1 case -pick

    for(int ind = 1; ind<n; ind++){
        for(int target= 0; target<=tar; target++){

            int notTaken = dp[ind-1][target];

            int taken = 0;
                if(num[ind]<=target)
                    taken = dp[ind-1][target-num[ind]];

            dp[ind][target]= (notTaken + taken)%mod;
        }
    }
    return dp[n-1][tar];
}
```

```
int targetSum(int n, int target, vector<int>& arr){
    int totSum = 0;
    for(int i=0; i<n;i++){
        totSum += arr[i];
    }

    //Checking for edge cases
    if(totSum-target <0 || (totSum-target)%2 ) return 0;

    return findWays(arr,(totSum-target)/2);
}



int main() {

  vector<int> arr = {1,2,3,1};
  int target=3;

  int n = arr.size();
  cout<<"The number of ways found is " <<targetSum(n,target,arr);
}
```

Output:

The number of ways found is 2

Time Complexity: O(N*K)

Reason: There are two nested loops

Space Complexity: O(N*K)

Reason: We are using an external array of size ‘N*K’. Stack Space is eliminated.

Part 3: Space Optimization

If we closely look the relation,

dp[ind][target] = dp[ind-1][target] + dp[ind-1][target-arr[ind]]

We see that to calculate a value of a cell of the dp array, we need only the previous row values (say prev). So, we don't need to store an entire array. Hence we can space optimize it.

Note: Whenever we create a new row (say cur), we need to explicitly set its first element as true according to our base condition.

Code:

- C++ Code
- Java Code

```cpp
#include <bits/stdc++.h>


using namespace std;


int mod =(int)1e9+7;


int findWays(vector<int> &num, int tar){
     int n = num.size();


    vector<int> prev(tar+1,0);


    if(num[0] == 0) prev[0] =2;  // 2 cases -pick and not pick
    else prev[0] = 1;  // 1 case - not pick

    if(num[0]!=0 && num[0]<=tar) prev[num[0]] = 1;  // 1 case -pick

    for(int ind = 1; ind<n; ind++){
        vector<int> cur(tar+1,0);
        for(int target= 0; target<=tar; target++){
            int notTaken = prev[target];

            int taken = 0;
                if(num[ind]<=target)
                    taken = prev[target-num[ind]];

            cur[target]= (notTaken + taken)%mod;
        }
        prev = cur;
    }
    return prev[tar];
}

int targetSum(int n, int target, vector<int>& arr){
    int totSum = 0;
    for(int i=0; i<n;i++){
        totSum += arr[i];
    }

    //Checking for edge cases
```

```
    if(totSum-target <0 || (totSum-target)%2 ) return 0;

    return findWays(arr,(totSum-target)/2);
}


int main() {

  vector<int> arr = {1,2,3,1};
  int n = arr.size();
  int target=3;

  cout<<"The number of subsets found is " <<targetSum(n,target,arr);
}
```

Output:

The number of ways found is 2

Time Complexity: O(N*K)

Reason: There are three nested loops

Space Complexity: O(K)

Reason: We are using an external array of size ‘K+1’ to store only one row.