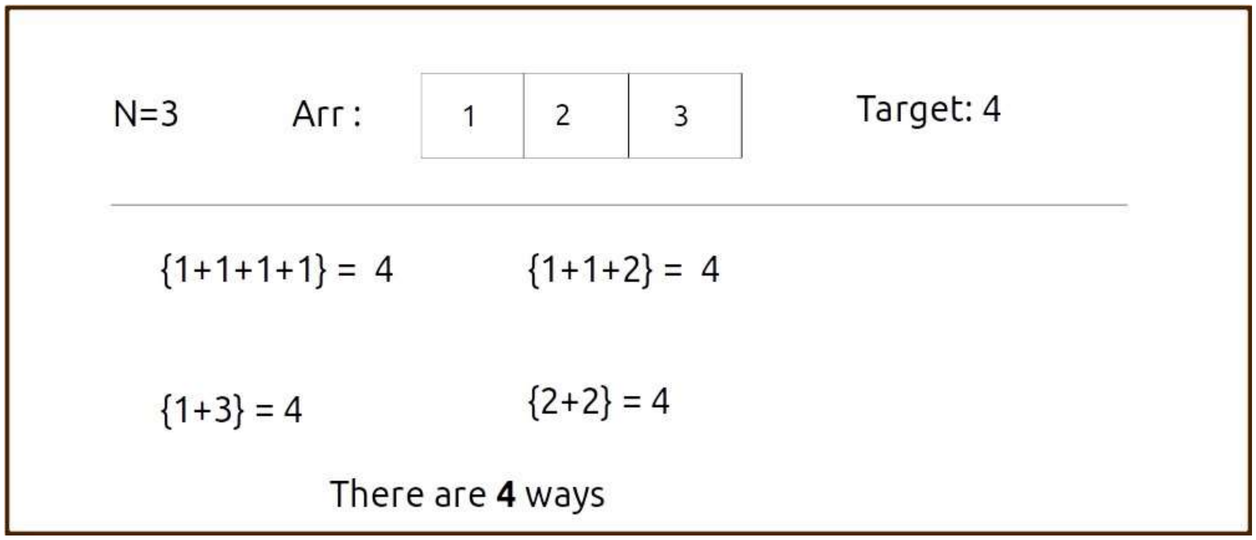
Coin Change 2 (DP – 22)

Problem Link: Ways to Make a Coin Change

We are given an array Arr with N distinct coins and a target. We have an infinite supply of each coin denomination. We need to find the number of ways we sum up the coin values to give us the target.

Each coin can be used any number of times.
`Example:`



Memorization:

Algorithm / Intuition

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.

We are given ‘n’ coins. Their denomination value is given by the array ‘arr’.So clearly one parameter will be ‘ind’, i.e index up to which the array items are being considered.

There is one more parameter, the given target value “T” which we want to achieve so that while generating subsequences, we can decide whether we want to include a particular coin or not.

So, we can say that initially, we need to find f(n-1, T) where T is the initial target given to us in the question. f(n-1, T) means we are finding the total number of ways to form the target T by considering coins from index 0 to index n-1 of the arr array.

f(ind,T) > Total number of ways, by considering coins from index 0 to ind-1, to get target T.

Base Cases:

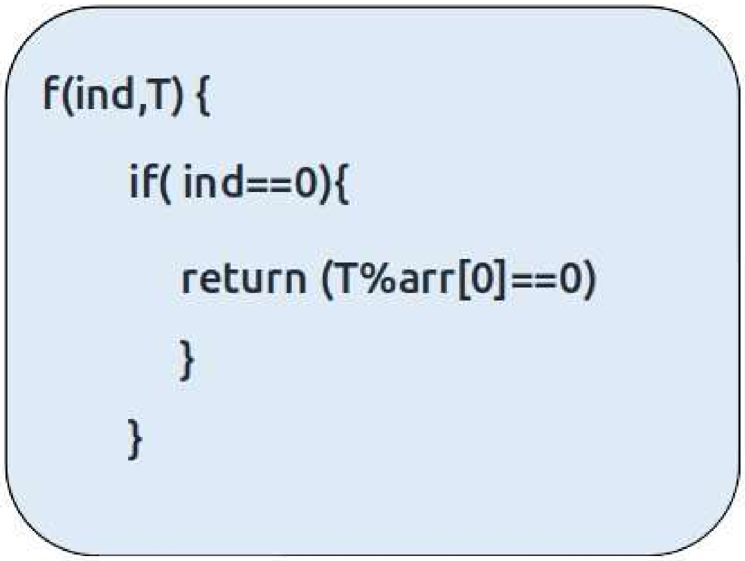
If ind==0, it means we are at the first item so we have only one coin denomination, therefore the following two cases can arise:

- **T is divisible by arr[0] (eg: arr[0] = 4** and **T = 12)**

In such a case where the target is divisible by the coin element value, we will return 1 as we will be able to form the target.

- **T is not divisible by arr[0] (eg: arr[0] = 4** and **T = 7)**

In all other cases, we will not be able to form the target, so we will return 0.



```
f(ind,T) {
      if( ind==0){
          return (T%arr[0]==0)
          }
      }
```

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

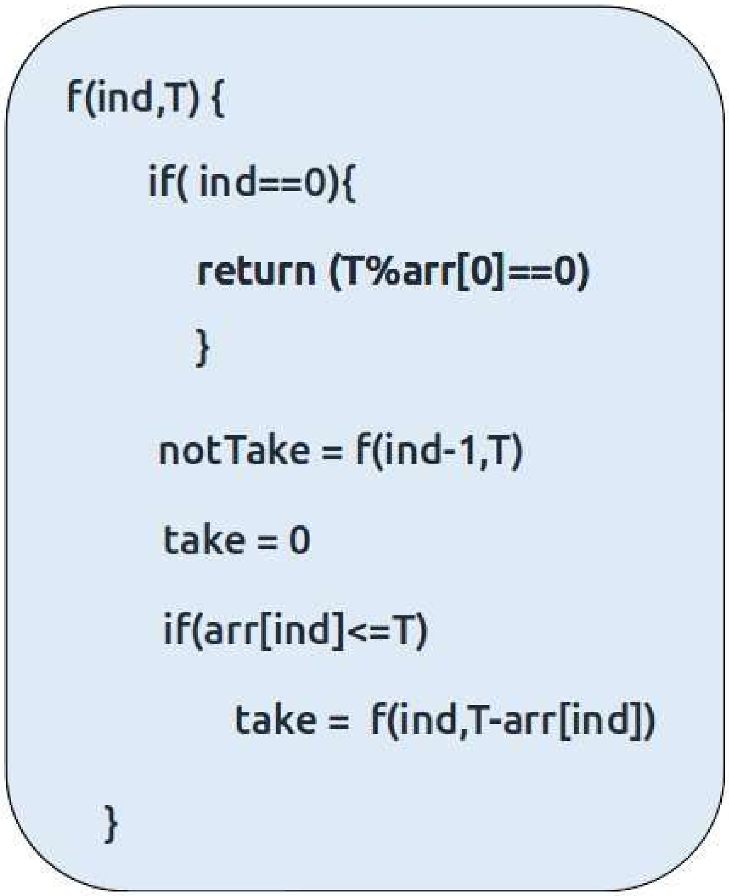
We need to generate all the subsequences. We will use the pick/non-pick technique as discussed in this video "Recursion on Subsequences".

We have two choices:

- **Exclude the current element in the subsequence:** We first try to find a subsequence without considering the current index coin. If we exclude the current coin, the target sum will not be affected. So we will call the recursive function **f(ind-1,T)** to find the remaining answer.
- **Include the current element in the subsequence:** We will try to find a subsequence by considering the current icoin. As we have included the coin, the target sum will be updated to T-arr[ind].

Now here is the catch, as there is an unlimited supply of coins, we want to again form a solution with the same coin value. So we **will not** recursively call for f(ind-1, T-arr[ind]) rather we will stay at that index only and call for f(find, **T-arr[ind])** to find the answer.
Note: We will consider the current coin only when its

denomination value (arr[ind]) is less than or equal to the target T.

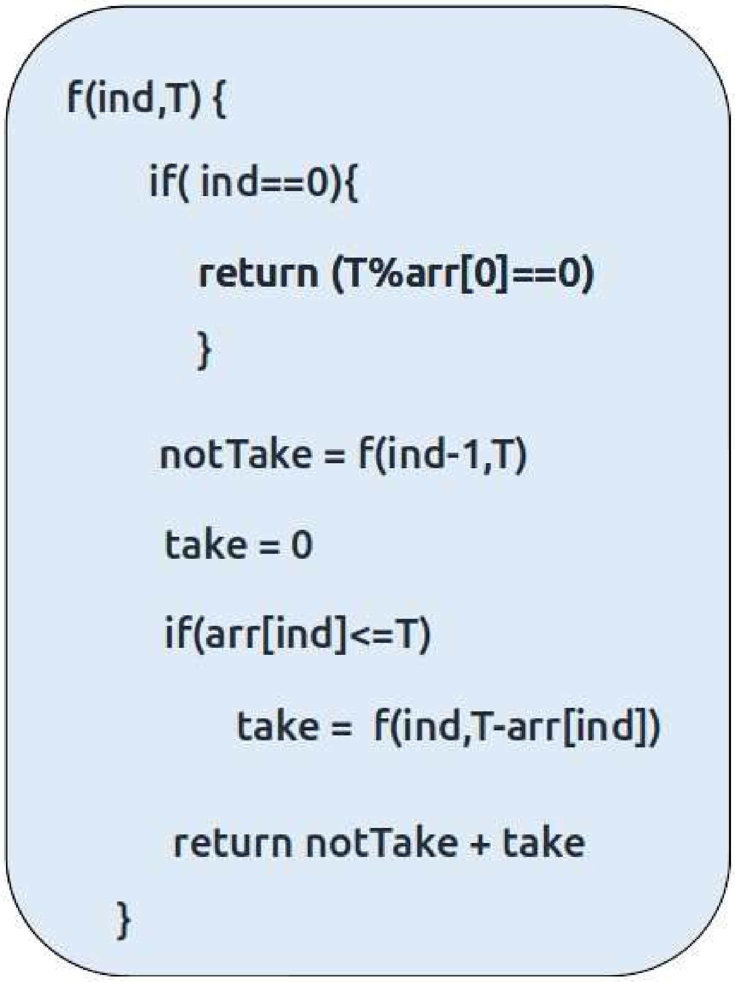


```
f(ind,T) {
      if( ind==0){
         return (T%arr[0]==0)
         }
      notTake = f(ind-1,T)
      take = 0
      if(arr[ind]<=T)
            take =  f(ind,T-arr[ind])
  }
```

Step 3: Return the sum of take and notTake

As we have to return the total number of ways we can form the target, we will return the sum of notTake and take as our answer.

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



```
f(ind,T) {
     if( ind==0){
        return (T%arr[0]==0)
        }
      notTake = f(ind-1,T)
      take = 0
      if(arr[ind]<=T)
           take =  f(ind,T-arr[ind])
       return notTake + take
  }
```

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 to the following steps will be taken:

1. Create a dp array of size [n][T+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 ‘T’. Therefore we take the dp array as dp[n][T+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.

```
#include <bits/stdc++.h>
using namespace std;

// Function to count the number of ways to make change for a
given target sum
long countWaysToMakeChangeUtil(vector<int>& arr, int ind, int
T, vector<vector<long>>& dp) {
   // Base case: if we're at the first element
   if (ind == 0) {
      // Check if the target sum is divisible by the first element
      return (T % arr[0] == 0);
   }

   // If the result for this index and target sum is already
calculated, return it
   if (dp[ind][T] != -1)
      return dp[ind][T];

   // Calculate the number of ways without taking the current
element
```

```
    long notTaken = countWaysToMakeChangeUtil(arr, ind - 1, T,
dp);

    // Calculate the number of ways by taking the current
element
    long taken = 0;
    if (arr[ind] <= T)
        taken = countWaysToMakeChangeUtil(arr, ind, T - arr[ind],
dp);

    // Store the sum of ways in the DP table and return it
    return dp[ind][T] = notTaken + taken;
}

// Function to count the number of ways to make change for
the target sum
long countWaysToMakeChange(vector<int>& arr, int n, int T) {
    vector<vector<long>> dp(n, vector<long>(T + 1, -1)); // Create
a DP table

    // Call the utility function to calculate the answer
    return countWaysToMakeChangeUtil(arr, n - 1, T, dp);
```

```
}

int main() {
    vector<int> arr = {1, 2, 3};
    int target = 4;
    int n = arr.size();

    cout << "The total number of ways is " <<
countWaysToMakeChange(arr, n, target) << endl;

    return 0; // Return 0 to indicate successful program execution
}
```

Output: The total number of ways is 4

Time Complexity: O(N*T)

Reason: There are N*W states therefore at max 'N*T' new problems will be solved.

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

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

Tabulation

Algorithm / Intuition

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.

- At ind==0, we are considering the first element, if the target value is divisible by the first coin's value, we set the cell's value as 1 or else 0.
- Next, we are done for the first row, so our 'ind' variable will move from 1 to n-1, whereas our 'target' variable will move from 0 to 'T'. We will set the nested loops to traverse the dp array.
- Inside the nested loops, we will apply the recursive logic to find the answer of the cell.
- When the nested loop execution has ended, we will return dp[n-1][T] as our answer.

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

using namespace std;


// Function to count the number of ways to make change for a
given target sum

long countWaysToMakeChange(vector<int>& arr, int n, int T) {

   vector<vector<long>> dp(n, vector<long>(T + 1, 0)); // Create
a DP table



   // Initializing base condition

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

      if (i % arr[0] == 0)
```

```
            dp[0][i] = 1;
        // Else condition is automatically fulfilled,
        // as dp array is initialized to zero
    }

    for (int ind = 1; ind < n; ind++) {
        for (int target = 0; target <= T; target++) {
            long notTaken = dp[ind - 1][target];

            long taken = 0;
            if (arr[ind] <= target)
                taken = dp[ind][target - arr[ind]];

            dp[ind][target] = notTaken + taken;
        }
    }

    return dp[n - 1][T];
}
```

```
int main() {
   vector<int> arr = {1, 2, 3};
   int target = 4;
   int n = arr.size();

   cout << "The total number of ways is " <<
countWaysToMakeChange(arr, n, target) << endl;

   return 0; // Return 0 to indicate successful program execution
}
```

Complexity Analysis

Time Complexity: O(N*T)

Reason: There are two nested loops

Space Complexity: O(N*T)

Reason: We are using an external array of size 'N*T'. Stack Space is eliminated.

Optimization:

Algorithm / Intuition

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: We first need to initialize the first row as we had done in the tabulation approach.

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

using namespace std;


// Function to count the number of ways to make change for a
given target sum

long countWaysToMakeChange(vector<int>& arr, int n, int T) {

   vector<long> prev(T + 1, 0);  // Create a vector to store the
previous DP state


   // Initialize base condition

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

      if (i % arr[0] == 0)

         prev[i] = 1;  // There is one way to make change for
multiples of the first coin

      // Else condition is automatically fulfilled,

      // as the prev vector is initialized to zero
```

```
  }


  for (int ind = 1; ind < n; ind++) {
      vector<long> cur(T + 1, 0);  // Create a vector to store the
current DP state
      for (int target = 0; target <= T; target++) {
          long notTaken = prev[target];  // Number of ways without
taking the current coin


          long taken = 0;
          if (arr[ind] <= target)
              taken = cur[target - arr[ind]];  // Number of ways by
taking the current coin


          cur[target] = notTaken + taken;  // Total number of ways
for the current target
      }
      prev = cur;  // Update the previous DP state with the
current state for the next coin
  }
```

```
    return prev[T];  // Return the total number of ways to make
change for the target
}

int main() {
    vector<int> arr = {1, 2, 3};  // Coin denominations
    int target = 4;  // Target sum
    int n = arr.size();  // Number of coin denominations

    // Call the function to calculate and output the total number
of ways to make change
    cout << "The total number of ways is " <<
countWaysToMakeChange(arr, n, target) << endl;

    return 0;  // Return 0 to indicate successful program
execution
}
```

Complexity Analysis

Time Complexity: O(N*T)

Reason: There are two nested loops.

Space Complexity: O(T)

Reason: We are using an external array of size ‘T+1’ to store two rows only.