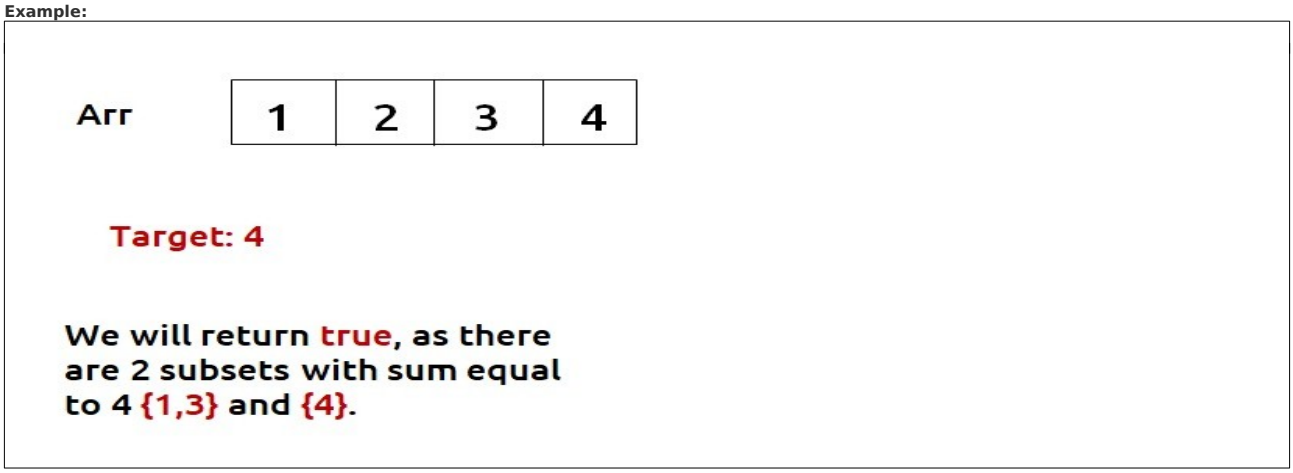
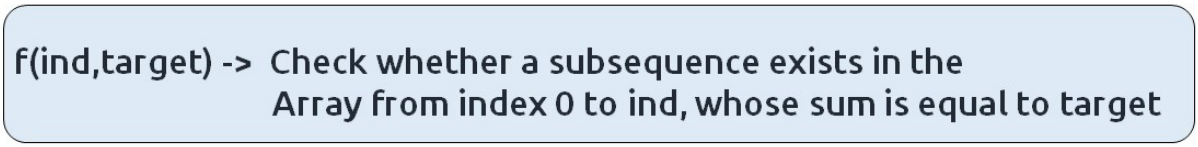
Subset sum equal to target (DP- 14)
In this article, we will solve the most asked coding interview problem: Subset sum equal to target.
In this article, we will be going to understand the pattern of dynamic programming on subsequences of an array. We will be using the problem "Subset Sum Equal to K".
First, we need to understand **what a subsequence/subset is**.
A subset/subsequence is a contiguous or non-contiguous part of an array, where elements appear in the same order as the original array.
For example, for the array: [2,3,1] , the subsequences will be [{2},{3},{1},{2,3},{2,1},{3,1},{2,3,1}} but {3,2} is **not** a subsequence because its elements are not in the same order as the original array.
Problem Link: Subset Sum Equal to K
We are given an array 'ARR' with N positive integers. We need to find if there is a subset in "ARR" with a sum equal to K. If there is, return true else return false.
Example:



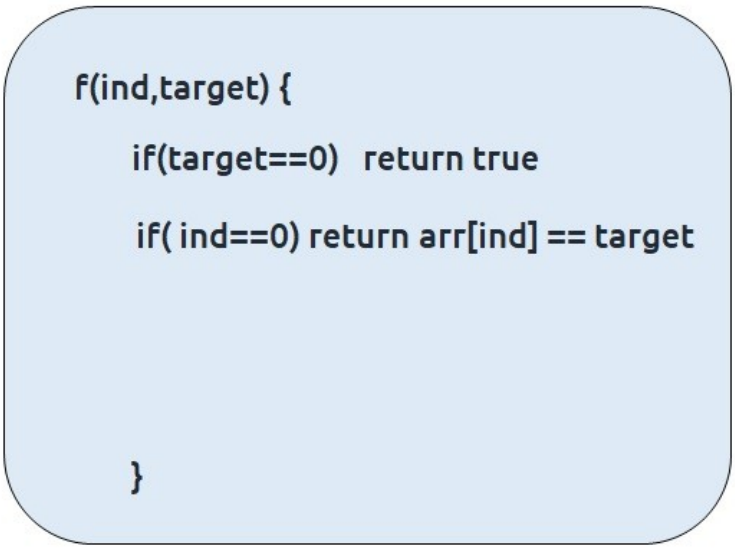
Memorization Approach:
Why a Greedy Solution doesn't work?
A Greedy Solution doesn't make sense because we are not looking to optimize anything. We can rather try to generate all subsequences using recursion and whenever we get a single subsequence whose sum is equal to the given target, we can return true.
Note: Readers are highly advised to watch this video "Recursion on Subsequences" to understand how we generate subsequences using recursion.

Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in the 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 need to find whether there exists a subsequence 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:



Base Cases:
- If target == 0, it means that we have already found the subsequence from the previous steps, so we can return true.
- 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 true else false.



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 or equal to the target.

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

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

}
```

Step 3: Return (taken || notTaken)
As we are looking for only one subset, if any of the one among taken or not taken returns true, we can return true from our function. Therefore, we return 'or(||)' of both of them.
The final pseudocode after steps 1, 2, and 3:

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

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

    return notTaken || taken
}
```

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.

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

// Function to check if there is a subset of 'arr' with a sum equal to 'target'
bool subsetSumUtil(int ind, int target, vector<int>& arr, vector<vector<int>>& dp) {
    // If the target sum is 0, we have found a subset
    if (target == 0)
        return true;

    // If we have reached the first element in 'arr'
    if (ind == 0)
        return arr[0] == target;

    // If the result for this subproblem has already been computed, return it
    if (dp[ind][target] != -1)
        return dp[ind][target];

    // Try not taking the current element into the subset
    bool notTaken = subsetSumUtil(ind - 1, target, arr, dp);

    // Try taking the current element into the subset if it doesn't exceed the target
    bool taken = false;
    if (arr[ind] <= target)
        taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);

    // Store the result in the dp array to avoid recomputation
    return dp[ind][target] = notTaken || taken;
}

// Function to check if there is a subset of 'arr' with a sum equal to 'k'
bool subsetSumToK(int n, int k, vector<int>& arr) {
    // Initialize a 2D DP array for memoization
    vector<vector<int>> dp(n, vector<int>(k + 1, -1));

    // Call the recursive subsetSumUtil function
    return subsetSumUtil(n - 1, k, arr, dp);
}

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

    // Call the subsetSumToK function and print the result
    if (subsetSumToK(n, k, arr))
        cout << "Subset with the given target found";
    else
        cout << "Subset with the given target not found";

    return 0;
}
```

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)).

Tabulation Approach:

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 set its type as bool and initialize it as false.

Eg: [4,2,3,7,...,23]

Target: 11

target →

ind ↓

target / ind	0	1	2	3	4	. . .	K
0	false	false	false	false	false	false	false
1	false	false	false	false	false	false	false
.	false	false	false	false	false	false	false
.							
.							
N-1	false	false	false	false	false	false	false

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 true.

Eg: [4,2,3,7,...,23]

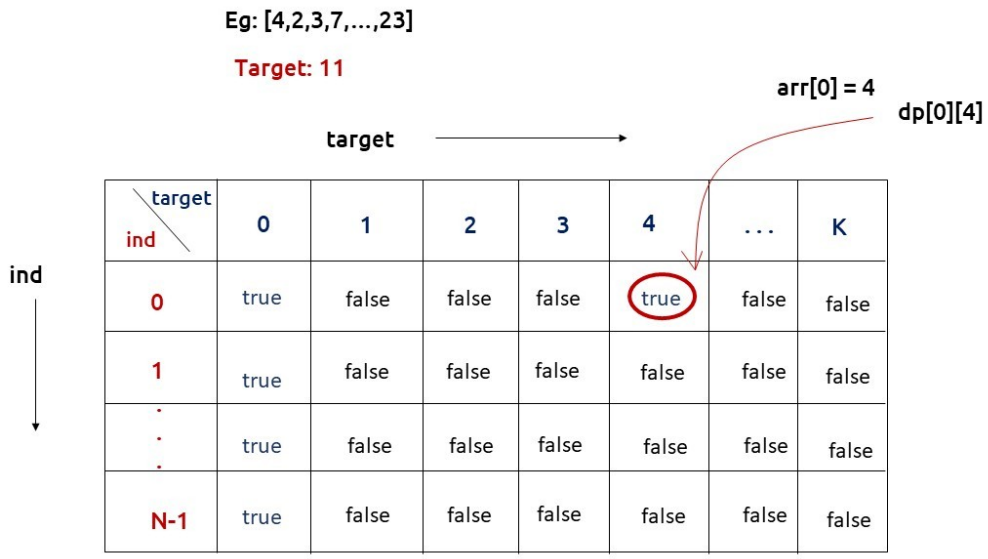
Target: 11

target →

ind ↓

target / ind	0	1	2	3	4	. . .	K
0	true	false	false	false	false	false	false
1	true	false	false	false	false	false	false
.	true	false	false	false	false	false	false
.							
.							
N-1	true	false	false	false	false	false	false

- 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]] =true, (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]] = true.



●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.

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

// Function to check if there is a subset of 'arr' with a sum equal to 'k'
bool subsetSumToK(int n, int k, vector<int> &arr) {
    // Initialize a 2D DP array with dimensions (n x k+1) to store subproblem results
    vector<vector<bool>> dp(n, vector<bool>(k + 1, false));

    // Base case: If the target sum is 0, we can always achieve it by taking no elements
    for (int i = 0; i < n; i++) {
        dp[i][0] = true;
    }

    // Base case: If the first element of 'arr' is less than or equal to 'k', set dp[0][arr[0]] to true
    if (arr[0] <= k) {
        dp[0][arr[0]] = true;
    }

    // Fill the DP array iteratively
    for (int ind = 1; ind < n; ind++) {
        for (int target = 1; target <= k; target++) {
            // If we don't take the current element, the result is the same as the previous row
            bool notTaken = dp[ind - 1][target];

            // If we take the current element, subtract its value from the target and check the previous row
            bool taken = false;
            if (arr[ind] <= target) {
                taken = dp[ind - 1][target - arr[ind]];
            }

            // Store the result in the DP array for the current subproblem
            dp[ind][target] = notTaken || taken;
        }
    }

    // The final result is stored in dp[n-1][k]
    return dp[n - 1][k];
}

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

    // Call the subsetSumToK function and print the result
    if (subsetSumToK(n, k, arr))
        cout << "Subset with the given target found";
    else
        cout << "Subset with the given target not found";

    return 0;
}
```

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.

Space Optimization Approach:

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 is true according to our base condition.

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

// Function to check if there is a subset of 'arr' with a sum equal to 'k'
bool subsetSumToK(int n, int k, vector<int> &arr) {
    // Initialize a vector 'prev' to store the previous row of the DP table
    vector<bool> prev(k + 1, false);

    // Base case: If the target sum is 0, we can always achieve it by taking no elements
    prev[0] = true;

    // Base case: If the first element of 'arr' is less than or equal to 'k', set prev[arr[0]] to true
    if (arr[0] <= k) {
        prev[arr[0]] = true;
    }

    // Iterate through the elements of 'arr' and update the DP table
    for (int ind = 1; ind < n; ind++) {
        // Initialize a new row 'cur' to store the current state of the DP table
        vector<bool> cur(k + 1, false);

        // Base case: If the target sum is 0, we can achieve it by taking no elements
        cur[0] = true;

        for (int target = 1; target <= k; target++) {
            // If we don't take the current element, the result is the same as the previous row
            bool notTaken = prev[target];

            // If we take the current element, subtract its value from the target and check the previous row
            bool taken = false;
            if (arr[ind] <= target) {
                taken = prev[target - arr[ind]];
            }

            // Store the result in the current DP table row for the current subproblem
            cur[target] = notTaken || taken;
        }

        // Update 'prev' with the current row 'cur' for the next iteration
        prev = cur;
    }

    // The final result is stored in prev[k]
    return prev[k];
}

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

    // Call the subsetSumToK function and print the result
    if (subsetSumToK(n, k, arr))
        cout << "Subset with the given target found";
    else
        cout << "Subset with the given target not found";

    return 0;
}
```

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.