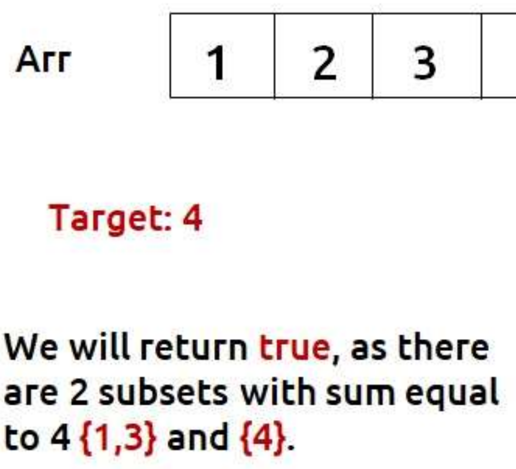
Subset sum equal to target (DP- 14)

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:



We will return true, as there are 2 subsets with sum equal to 4 {1,3} and {4}.

Pre-req: Dynamic Programming Introduction, Recursion on Subsequences

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

Solution :

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.

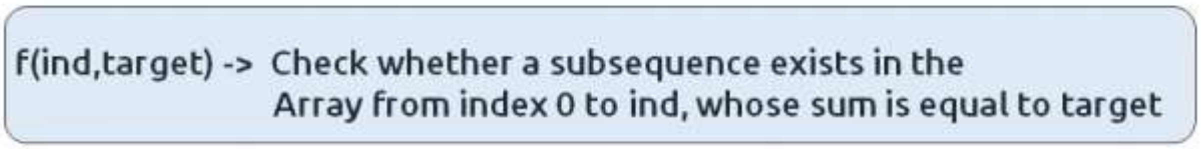
Steps to form the recursive solution:

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.

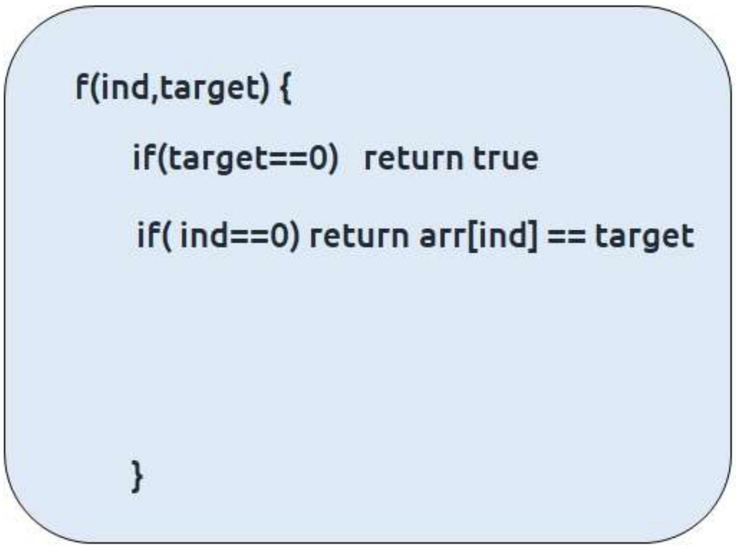
So, we can say that initially, we need to find(n-1, target) which means that we need to find whether there exists a subsequence in the array from index 0 to n-1, whose sum is equal to the target. Similarly, we can generalize it for any index ind as follows:



f(ind,target) -> Check whether a subsequence 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 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.



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



}
```

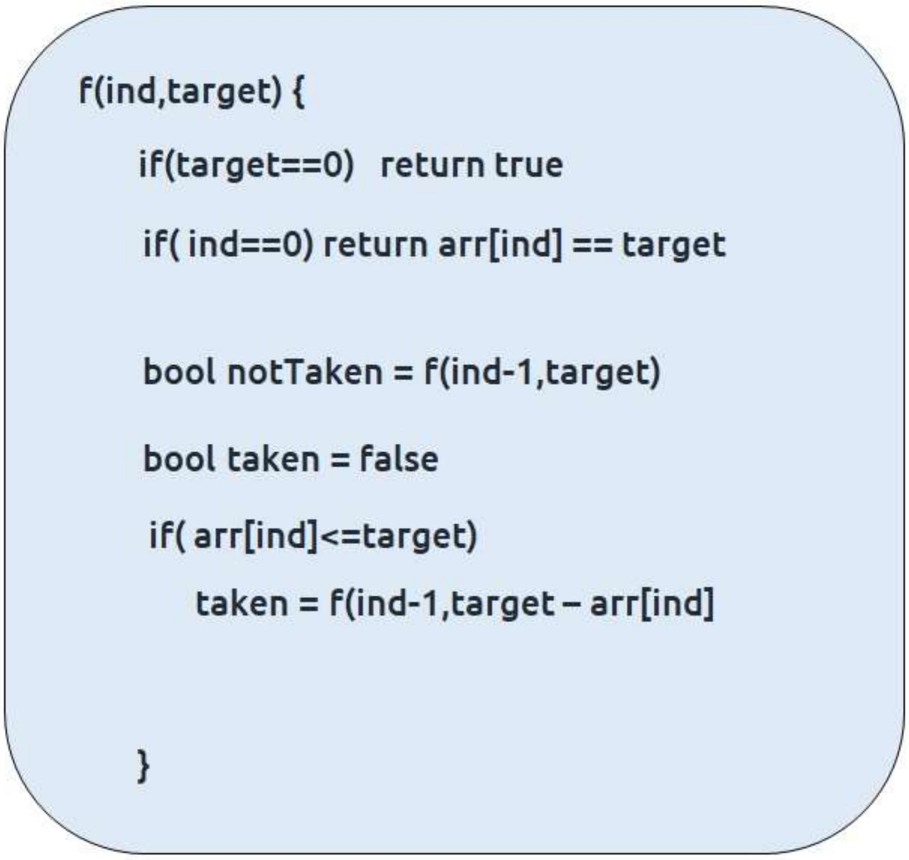
Step 2: Try out all possible choices at a given index.

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 or equal to the target.

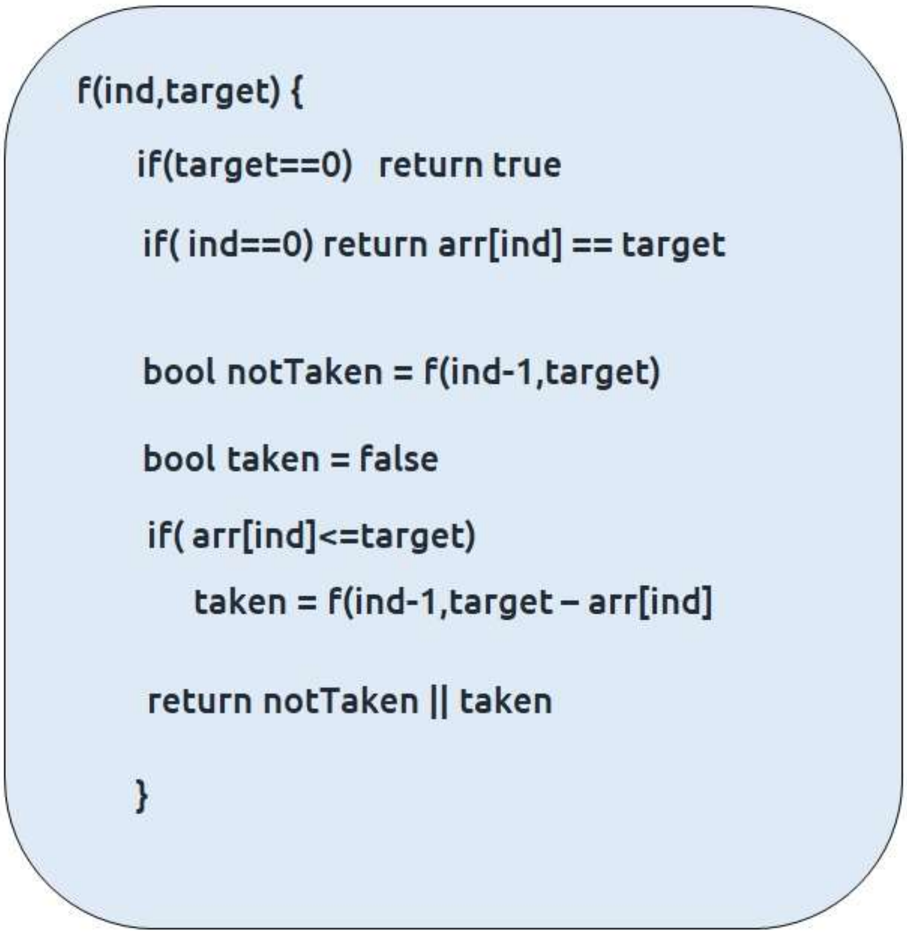


```
f(ind,target) {
    if(target==0)   return true
    if( ind==0) return arr[ind] == target

    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.

Code:

- C++ Code

- Java Code

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

using namespace std;

bool subsetSumUtil(int ind, int target, vector<int>& arr, vector<vector<int>>
&dp){
    if(target==0)
        return true;

    if(ind == 0)
        return arr[0] == target;

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

    bool notTaken = subsetSumUtil(ind-1,target,arr,dp);

    bool taken = false;
    if(arr[ind]<=target)
        taken = subsetSumUtil(ind-1,target-arr[ind],arr,dp);

    return dp[ind][target]= notTaken||taken;
}

bool subsetSumToK(int n, int k, vector<int> &arr){
    vector<vector<int>> dp(n,vector<int>(k+1,-1));

    return subsetSumUtil(n-1,k,arr,dp);
}

int main() {

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

  if(subsetSumToK(n,k,arr))
    cout<<"Subset with given target found";
  else
    cout<<"Subset with given target not found";
}
```

Output:

Subset with given target found

Time Complexity: O(N*K)

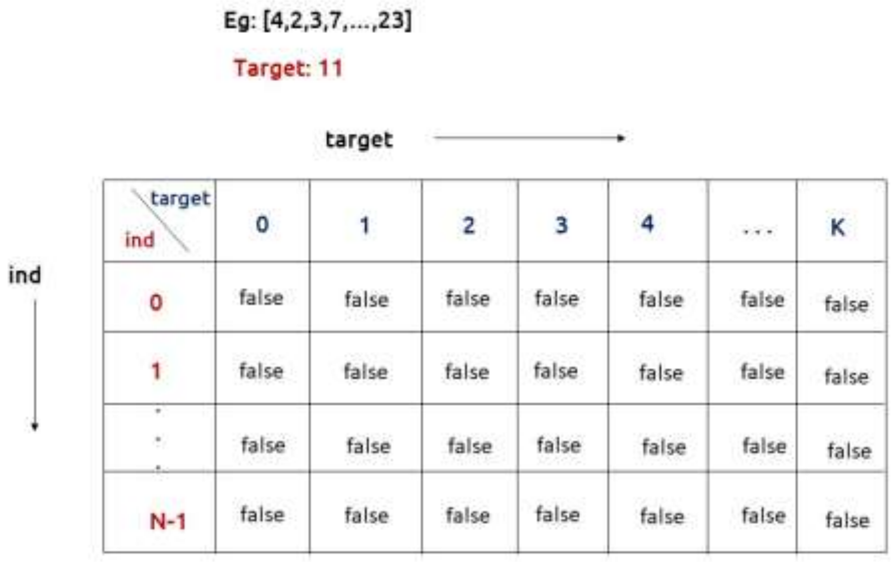
Reason: There are N*K states therefore at max ‘N*K’ new problems will be solved.

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 set its type as bool and initialize it as false.


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

Target: 11

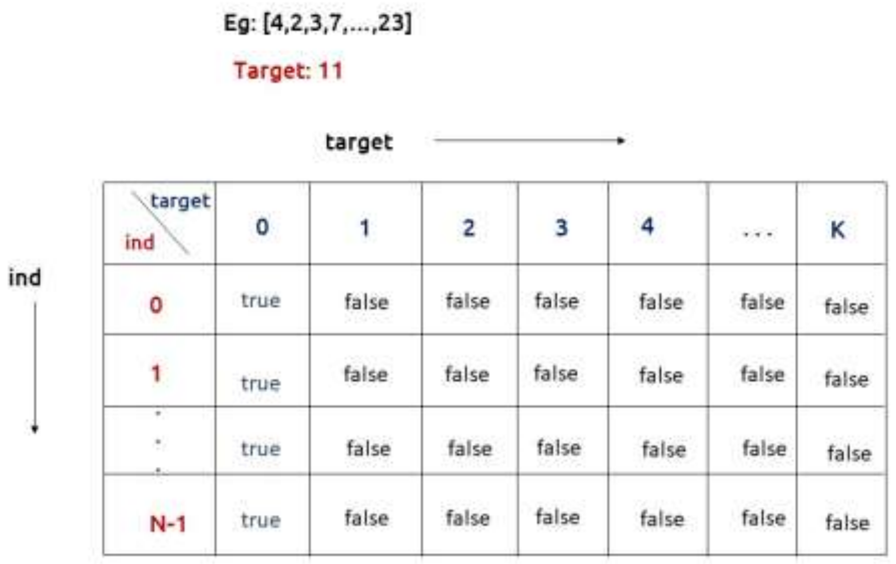
target →

ind ↓

ind \ target	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.

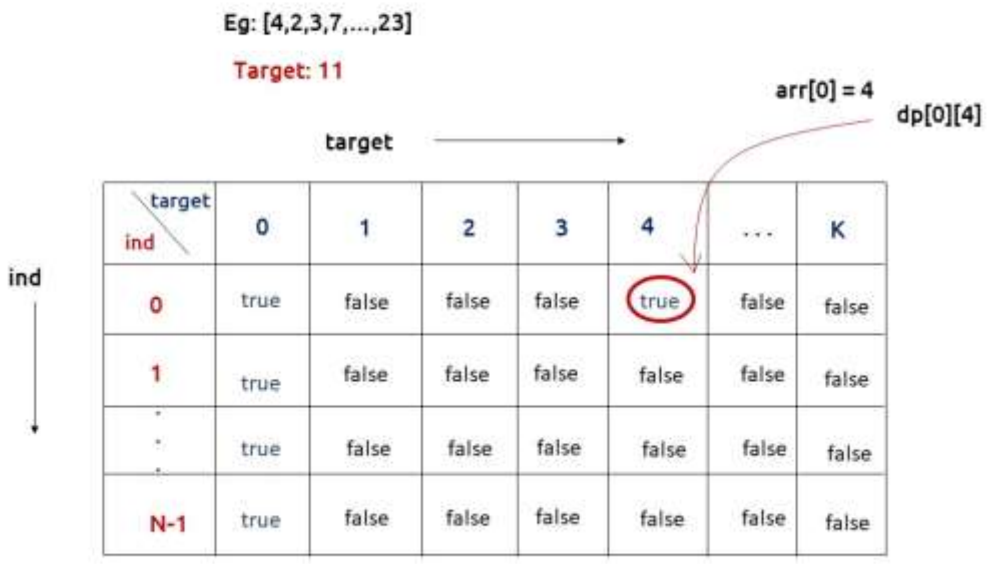
- If target == 0, ind can take any value from 0 to n-1, therefore we need to set the value of the first column as true.


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

Target: 11

ind \ target	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.


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

Target: 11

arr[0] = 4
dp[0][4]
target
ind

ind \ target	0	1	2	3	4	...	K
0	true	false	false	false	true	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

- 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;

bool subsetSumToK(int n, int k, vector<int> &arr){
    vector<vector<bool>> dp(n,vector<bool>(k+1,false));

    for(int i=0; i<n; i++){
        dp[i][0] = true;
    }

    if(arr[0]<=k)
        dp[0][arr[0]] = true;

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

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

            bool taken = false;
                if(arr[ind]<=target)
                    taken = dp[ind-1][target-arr[ind]];

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

    return dp[n-1][k];
}

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

  if(subsetSumToK(n,k,arr))
    cout<<"Subset with given target found";
  else
    cout<<"Subset with given target not found";
}
```

Output:

Subset with given target found

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 is true according to our base condition.

Code:

- C++ Code
- Java Code

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


using namespace std;
```

```cpp
bool subsetSumToK(int n, int k, vector<int> &arr){
    vector<bool> prev(k+1,false);

    prev[0] = true;

    if(arr[0]<=k)
        prev[arr[0]] = true;

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

            bool taken = false;
                if(arr[ind]<=target)
                    taken = prev[target-arr[ind]];

            cur[target]= notTaken||taken;
        }
        prev = cur;
    }

    return prev[k];
}

int main() {

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

  if(subsetSumToK(n,k,arr))
    cout<<"Subset with given target found";
  else
    cout<<"Subset with given target not found";
}
```

Output:

Subset with given target found

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.