Subset Sum Equals K

Author:	Prakash JC

Problem Statement

Given an array of integers and an integer `k`, determine whether there exists a subset whose sum is equal to `k`.

Editorial

General ways to do is Power set and Recursion.

Instead of generating all subsets, we just need to check whether there is a subset with sum equal to k.

So, one way is, do recursion, and when we get one subset, stop there.

Recursion Approach

States: `Index` of array and `target`.

Possibilities: An index can be a part or not a part of the subset.

Direction: `n-1` to `0`

Define the problem: Does there exist a sum with target in the entire array.

Base cases:

1. If target = 0.

```
if (target == 0) return true;
```

2. Since the direction is from n-1 to 0, if the reamaining target is same as the arr[0], then a subset exists.

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

Explore all paths:

At each index we have two choices, pick or not pick. Target will be reduced if we take a value.

Note: The `target` should be greater than or equal to the value, else we can't take it.

If *either one* of these paths leads to a valid subset, the answer should be `true`.

Note: We first declare `bool take = false;` so that the variable exists even if the `if` condition fails (`arr[index] > k`). It is not *strictly required.*

Code

```
bool f(int index, int k, vector<int>& arr) {
    if (k == 0) return true;
    if (index == 0) return arr[0] == k;

    bool notTake = f(index - 1, k, arr);

    bool take = false;
    if (arr[index] <= k)
        take = f(index - 1, k - arr[index], arr);
```

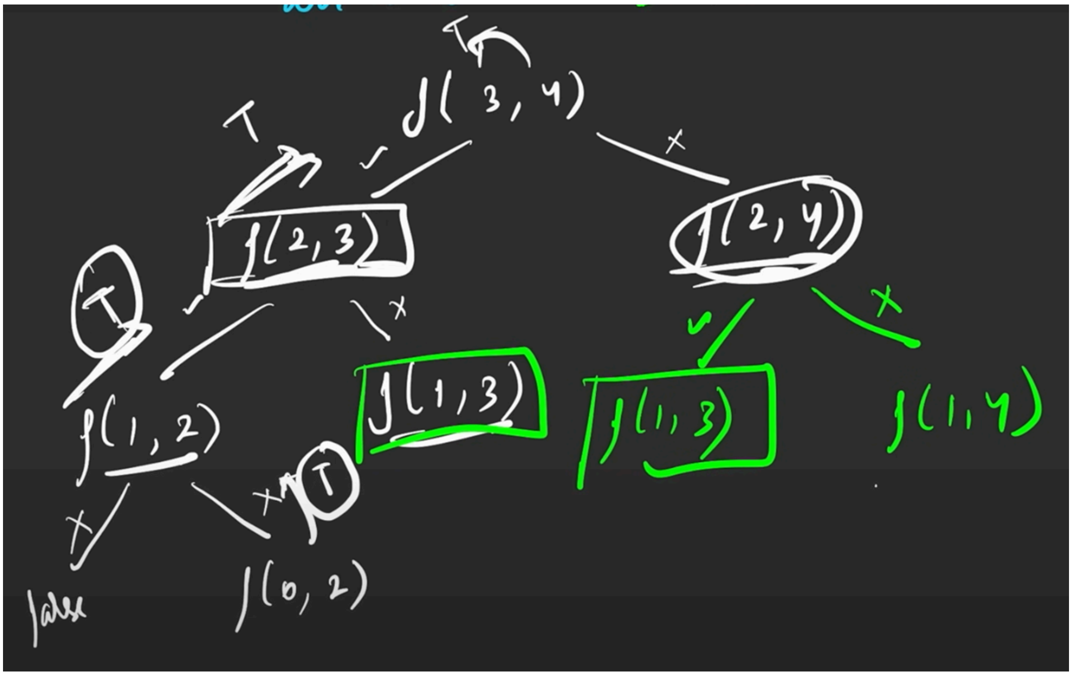
```
    return take || notTake;
}
```

Complexity Analysis

Time Complexity: `O(2^n)` – Each element has two choices (pick or not pick).

Space Complexity: `O(n)` – Recursive call stack depth at most `n`.

Draw the recursion tree to notice the overlapping subproblems:



Memoization

Approach

States: Index and target.

Use a **2D DP array** `dp[index][k]` where:

- `dp[i][target]` = true if it's possible to form sum `target` using elements `0..i`.
- If already calculated, return it.
- Otherwise, compute using recursion + store result.

Code

```
bool f(int index, int k, vector<int>& arr, vector<vector<int>>& dp) {
    if (k == 0) return true;
    if (index == 0) return arr[0] == k;
    if (dp[index][k] != -1) return dp[index][k];

    bool notTake = f(index - 1, k, arr, dp);
    bool take = false;
```

```
    if (arr[index] <= k)
        take = f(index - 1, k - arr[index], arr, dp);

    return dp[index][k] = (take || notTake);
}
```

Complexity Analysis

Time Complexity: `O(n * k)` – Each state `(index, k)` is solved once.

Space Complexity: `O(n * k)` + `O(n)` recursion stack.

Tabulation

Approach

Declare a `bool` dp array.

Initialization:

Look at the prev base cases:

1. For any index, if the target is 0, return true i.e., `dp[i][0] = true;`
2. For index = 0, return true only if the target value is equal to arr[0] i.e., `dp[0][a[0]] = true;`

Transition:

Bottom-Up: Loop index from 0 to n-1 and target from 0 to target.

Tip: Then paste the memoization solution and change the f to dp.

Clear intuition:

For each index, we have 2 choices: not take or take.

1 Not take:

The answer to the question "does there exist a subset with `current target` from `0` till `current index` if we **don't take** the current element" lies on the answer block of the question "does there exist a subset with the same `current target` **but** from `0` till `current index - 1` i.e., in `dp[i-1][t]`.

Similarly,

2 Take: Check if `target - arr[i]` was achievable with earlier elements (`dp[i-1][target - arr[i]]`).

Finally, take the OR of them and fill the value:

```
dp[i][t] = dp[i-1][t] || (t >= arr[i] && dp[i-1][t - arr[i]]).
```

Code

```
bool f(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 i = 1; i < n; i++) {
        for (int target = 1; target <= k; target++) {
```

```
            bool notTake = dp[i-1][target];
            bool take = false;
            if (arr[i] <= target) take = dp[i-1][target - arr[i]];
            dp[i][target] = take || notTake;
        }
    }
    return dp[n-1][k];
}
```

Complexity Analysis

Time Complexity: `O(n * k)` – Iterating over `n * k` table.

Space Complexity: `O(n * k)` – Full DP table.

Space Optimization

Approach

Notice: `dp[i][*]` depends only on `dp[i-1][*]`.

So we can keep just **two 1D arrays** (previous and current), or even reduce to **one array updated in reverse**.

Initialization:

Note: `prev[0] = true;` and **always** `curr[0] = true;` because if target is 0, answer is true.

Also, `prev[arr[0]] = true;`

Transition:

Rename dp[i-1] with prev and dp[i] with curr.

Do prev = curr before moving to the next row.

Code

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

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

    for (int i = 1; i < n; i++) {
        curr[0] = true;
        for (int target = 1; target <= k; target++) {
            bool notTake = prev[target];
            bool take = false;
            if (arr[i] <= target) take = prev[target - arr[i]];
            curr[target] = take || notTake;
        }
        prev = curr;
    }
    return prev[k];
}
```

Complexity Analysis

Time Complexity: `O(n * k)` – Same number of transitions.

Space Complexity: `O(k)` – Only one row stored at a time.

THE END