**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jEPRo5PDvx#problem-statement)

Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both the subsets is equal.

**Example 1:**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jEPRo5PDvx#example-1)

Input: {1, 2, 3, 4}  
Output: True  
Explanation: The given set can be partitioned into two subsets with equal sum: {1, 4} & {2, 3}

**Example 2:**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jEPRo5PDvx#example-2)

Input: {1, 1, 3, 4, 7}  
Output: True  
Explanation: The given set can be partitioned into two subsets with equal sum: {1, 3, 4} & {1, 7}

**Example 3:**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jEPRo5PDvx#example-3)

Input: {2, 3, 4, 6}  
Output: False  
Explanation: The given set cannot be partitioned into two subsets with equal sum.

Assume if S represents the total sum of all the given numbers, then the two equal subsets must have a sum equal to S/2. This essentially transforms our problem to: "Find a subset of the given numbers that has a total sum of S/2".

### Top-down Dynamic Programming with Memoization [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jEPRo5PDvx#top-down-dynamic-programming-with-memoization)

We can use memoization to overcome the overlapping sub-problems. As stated in previous lessons, memoization is when we store the results of all the previously solved sub-problems return the results from memory if we encounter a problem that has already been solved.

Since we need to store the results for every subset and for every possible sum, therefore we will be using a two-dimensional array to store the results of the solved sub-problems. The **first dimension** of the array will represent different subsets and the **second dimension** will represent different ‘sums’ that we can calculate from each subset. These two dimensions of the array can also be inferred from the two changing values (sum and currentIndex) in our recursive function canPartitionRecursive().

public boolean canPartition(int[] num) {

    int sum = 0;

    for (int i = 0; i < num.length; i++)

      sum += num[i];

    // if 'sum' is a an odd number, we can't have two subsets with equal sum

    if (sum % 2 != 0)

      return false;

    Boolean[][] dp = new Boolean[num.length][sum / 2 + 1];

    return this.canPartitionRecursive(dp, num, sum / 2, 0);

  }

  private boolean canPartitionRecursive(Boolean[][] dp, int[] num, int sum, int currentIndex) {

    // base check

    if (sum == 0)

      return true;

    if (num.length == 0 || currentIndex >= num.length)

      return false;

    // if we have not already processed a similar problem

    if (dp[currentIndex][sum] == null) {

      // recursive call after choosing the number at the currentIndex

      // if the number at currentIndex exceeds the sum, we shouldn't process this

      if (num[currentIndex] <= sum) {

        if (canPartitionRecursive(dp, num, sum - num[currentIndex], currentIndex + 1)) {

          dp[currentIndex][sum] = true;

          return true;

        }

      }

      // recursive call after excluding the number at the currentIndex

      dp[currentIndex][sum] = canPartitionRecursive(dp, num, sum, currentIndex + 1);

    }

    return dp[currentIndex][sum];

  }

### Bottom-up Dynamic Programming [#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/3jEPRo5PDvx#bottom-up-dynamic-programming)

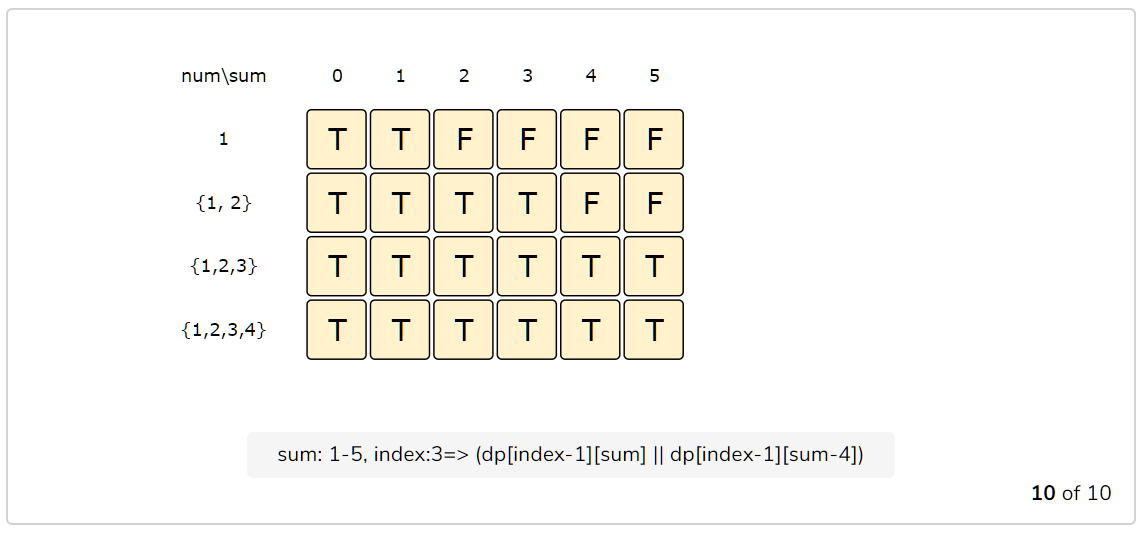
Let’s try to populate our dp[][] array from the above solution, working in a bottom-up fashion. Essentially, we want to find if we can make all possible sums with every subset. **This means, dp[i][s] will be ‘true’ if we can make sum ‘s’ from the first ‘i’ numbers.**

So, for each number at index ‘i’ (0 <= i < num.length) and sum ‘s’ (0 <= s <= S/2), we have two options:

1. Exclude the number. In this case, we will see if we can get ‘s’ from the subset excluding this number: dp[i-1][s]
2. Include the number if its value is not more than ‘s’. In this case, we will see if we can find a subset to get the remaining sum: dp[i-1][s-num[i]]

If either of the two above scenarios is true, we can find a subset of numbers with a sum equal to ‘s’.

Let’s start with our base case of zero capacity:



public boolean canPartition(int[] num) {

    int n = num.length;

    // find the total sum

    int sum = 0;

    for (int i = 0; i < n; i++)

      sum += num[i];

    // if 'sum' is a an odd number, we can't have two subsets with same total

    if(sum % 2 != 0)

      return false;

    // we are trying to find a subset of given numbers that has a total sum of ‘sum/2’.

    sum /= 2;

    boolean[][] dp = new boolean[n][sum + 1];

    // populate the sum=0 column, as we can always have '0' sum without including any element

    for(int i=0; i < n; i++)

      dp[i][0] = true;

    // with only one number, we can form a subset only when the required sum is equal to its value

    for(int s=1; s <= sum ; s++) {

      dp[0][s] = (num[0] == s ? true : false);

    }

    // process all subsets for all sums

    for(int i=1; i < n; i++) {

      for(int s=1; s <= sum; s++) {

        // if we can get the sum 's' without the number at index 'i'

        if(dp[i-1][s]) {

          dp[i][s] = dp[i-1][s];

        } else if (s >= num[i]) { // else if we can find a subset to get the remaining sum

          dp[i][s] = dp[i-1][s-num[i]];

        }

      }

    }

    // the bottom-right corner will have our answer.

    return dp[n-1][sum];

  }

The above solution has time and space complexity of O(N\*S)*O*(*N*∗*S*), where ‘N’ represents total numbers and ‘S’ is the total sum of all the numbers.

The 1-D implementation in Github repo.