**Minimum Subset Sum Difference**

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

Given a set of positive numbers, partition the set into two subsets with a minimum difference between their subset sums.

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

Input: {1, 2, 3, 9}  
Output: 3  
Explanation: We can partition the given set into two subsets where minimum absolute difference   
between the sum of numbers is '3'. Following are the two subsets: {1, 2, 3} & {9}.

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

Input: {1, 2, 7, 1, 5}  
Output: 0  
Explanation: We can partition the given set into two subsets where minimum absolute difference   
between the sum of number is '0'. Following are the two subsets: {1, 2, 5} & {7, 1}.

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

Input: {1, 3, 100, 4}  
Output: 92  
Explanation: We can partition the given set into two subsets where minimum absolute difference   
between the sum of numbers is '92'. Here are the two subsets: {1, 3, 4} & {100}.

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

We can use memoization to overcome the overlapping sub-problems.

We will be using a two-dimensional array to store the results of the solved sub-problems. We can uniquely identify a sub-problem from ‘currentIndex’ and ‘Sum1’; as ‘Sum2’ will always be the sum of the remaining numbers.

#### Code [**#**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/xVVNRPPXQGr#code-2)

Here is the code:

class PartitionSet {

  public int canPartition(int[] num) {

    int sum = 0;

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

      sum += num[i];

    Integer[][] dp = new Integer[num.length][sum + 1];

    return this.canPartitionRecursive(dp, num, 0, 0, 0);

  }

  private int canPartitionRecursive(Integer[][] dp, int[] num, int currentIndex, int sum1, int sum2) {

    // base check

    if(currentIndex == num.length)

      return Math.abs(sum1 - sum2);

    // check if we have not already processed similar problem

    if(dp[currentIndex][sum1] == null) {

      // recursive call after including the number at the currentIndex in the first set

      int diff1 = canPartitionRecursive(dp, num, currentIndex + 1, sum1 + num[currentIndex], sum2);

      // recursive call after including the number at the currentIndex in the second set

      int diff2 = canPartitionRecursive(dp, num, currentIndex + 1, sum1, sum2 + num[currentIndex]);

      dp[currentIndex][sum1] = Math.min(diff1, diff2);

    }

    return dp[currentIndex][sum1];

  }

  public static void main(String[] args) {

    PartitionSet ps = new PartitionSet();

    int[] num = {1, 2, 3, 9};

    System.out.println(ps.canPartition(num));

    num = new int[]{1, 2, 7, 1, 5};

    System.out.println(ps.canPartition(num));

    num = new int[]{1, 3, 100, 4};

    System.out.println(ps.canPartition(num));

  }

}

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

Let’s assume ‘S’ represents the total sum of all the numbers. So what we are trying to achieve in this problem is to find a subset whose sum is as close to ‘S/2’ as possible, because if we can partition the given set into two subsets of an equal sum, we get the minimum difference i.e. zero. This transforms our problem to [Subset Sum](https://www.educative.io/collection/page/5668639101419520/5633779737559040/5646239437684736/), where we try to find a subset whose sum is equal to a given number-- ‘S/2’ in our case. If we can’t find such a subset, then we will take the subset which has the sum **closest** to ‘S/2’. This is easily possible, as we will be calculating all possible sums with every subset.

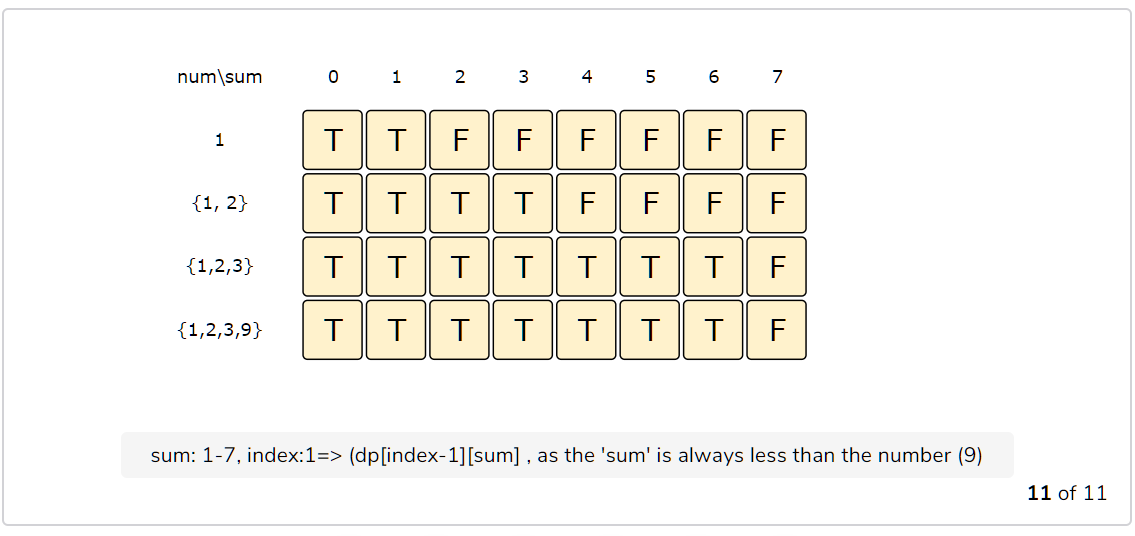
Essentially, we need to calculate all the possible sums up to ‘S/2’ for all numbers. So how do we populate the array db[TotalNumbers][S/2+1] in the bottom-up fashion?

For every possible sum ‘s’ (where 0 <= s <= S/2), we have two options:

1. Exclude the number. In this case, we will see if we can get the sum ‘s’ from the subset excluding this number => dp[index-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[index-1][s-num[index]]

If either of the two above scenarios is true, we can find a subset with a sum equal to ‘s’. We should dig into this before we can learn how to find the closest subset.

Let’s draw this visually, with the example input {1, 2, 3, 9}. Since the total sum is ‘15’, therefore, we will try to find a subset whose sum is equal to the half of it i.e. ‘7’.



class PartitionSet {

  public int canPartition(int[] num) {

    int sum = 0;

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

      sum += num[i];

    int n = num.length;

    boolean[][] dp = new boolean[n][sum/2 + 1];

    // populate the sum=0 columns, as we can always form '0' sum with an empty set

    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 that number

    for(int s=1; s <= sum/2 ; s++) {

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

    }

    // process all subsets for all sums

    for(int i=1; i < num.length; i++) {

      for(int s=1; s <= sum/2; 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 include the number and see if we can find a subset to get the remaining sum

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

        }

      }

    }

    int sum1 = 0;

    // Find the largest index in the last row which is true

    for (int i = sum / 2; i >= 0; i--) {

      if (dp[n-1][i] == true) {

          sum1 = i;

          break;

      }

    }

    int sum2 = sum - sum1;

    return Math.abs(sum2-sum1);

  }

  public static void main(String[] args) {

    PartitionSet ps = new PartitionSet();

    int[] num = {1, 2, 3, 9};

    System.out.println(ps.canPartition(num));

    num = new int[]{1, 2, 7, 1, 5};

    System.out.println(ps.canPartition(num));

    num = new int[]{1, 3, 100, 4};

    System.out.println(ps.canPartition(num));

  }

}