<https://leetcode.com/problems/partition-equal-subset-sum/description>/

Given a **non-empty** array nums containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

**Example 1:**

Input: nums = [1,5,11,5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

**Example 2:**

Input: nums = [1,2,3,5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.

**Constraints:**

* 1 <= nums.length <= 200
* 1 <= nums[i] <= 100

**Attempt 1: 2022-12-19**

**Solution 1:  Native DFS (TLE, 10 min)**

Basic Solution

This problem follows the 0/1 Knapsack pattern. A basic brute-force solution could be to try all combinations of partitioning the given numbers into two sets to see if any pair of sets has an 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".

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for(int num : nums) {

sum += num;

}

if(sum % 2 == 1) {

return false;

}

int target = sum / 2;

return helper(nums, target, 0);

}

private boolean helper(int[] nums, int target, int index) {

if(index >= nums.length) {

return false;

}

if(target == 0) {

return true;

}

// Pick current element

if(target >= nums[index]) {

if(helper(nums, target - nums[index], index + 1)) {

return true;

}

}

// Not pick current element

if(helper(nums, target, index + 1)) {

return true;

}

return false;

}

}

**Solution 2:  DFS with Memoization (Top down DP) (10 min)**

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 ‘target’ that we can calculate from each subset. These two dimensions of the array can also be inferred from the two changing values (target and index) in our recursive function helper(). The above algorithm has time and space complexity of O(N\*S), where ‘N’ represents total numbers and ‘S’ is the total sum of all the numbers.

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for(int num : nums) {

sum += num;

}

if(sum % 2 == 1) {

return false;

}

int target = sum / 2;

// No need store for index >= nums.length case, it means 'memo' will

// only store index < nums.length case, 1-dimension only need nums.length

// instead of 1 + nums.length and promote both TC and SC

//Boolean[][] memo = new Boolean[1 + nums.length][1 + target];

Boolean[][] memo = new Boolean[nums.length][1 + target];

return helper(nums, target, 0, memo);

}

private boolean helper(int[] nums, int target, int index, Boolean[][] memo) {

if(index >= nums.length) {

// No need store for index >= nums.length case, it means 'memo' will

// only store index < nums.length case, 1-dimension only need nums.length

// instead of 1 + nums.length and promote both TC and SC

//memo[index][target] = false;

return false;

}

if(memo[index][target] != null) {

return memo[index][target];

}

if(target == 0) {

memo[index][target] = true;

return true;

}

// Pick current element

if(target >= nums[index]) {

if(helper(nums, target - nums[index], index + 1, memo)) {

memo[index][target] = true;

return true;

}

}

// Not pick current element

if(helper(nums, target, index + 1, memo)) {

memo[index][target] = true;

return true;

}

memo[index][target] = false;

return false;

}

}

**Solution 3: 2D bottom up DP (10 min)**

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’.

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

**The big difference is happen on initialize first row, style 3's first row actually depends on index = 0 element on given array, style 1 & 2's first row is dummy row which has no element pick up from given array**

**Style 1: Initialize boolean[][] dp = new boolean[1 + nums.length][1 + target]**

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for(int num : nums) {

sum += num;

}

if(sum % 2 == 1) {

return false;

}

int target = sum / 2;

boolean[][] dp = new boolean[1 + nums.length][1 + target];

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

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

dp[i][0] = true;

}

// For len as 0(no number in nums array pick up),

// given any target, we can not make it

for(int i = 1; i <= target; i++) {

dp[0][i] = false;

}

// process all subsets for all sums

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

for(int j = 1; j <= target; j++) {

// if we can get the sum 'j' without the number at index 'i - 1'

if(dp[i - 1][j]) {

dp[i][j] = dp[i - 1][j];

// else if we can find a subset to get the remaining sum

} else if(j >= nums[i - 1]) {

dp[i][j] = dp[i - 1][j - nums[i - 1]];

}

}

}

return dp[nums.length][target];

}

}

**Style 2: Initialize boolean[][] dp = new boolean[1 + nums.length][1 + target]**

**1. Directly inherit boolean result get the sum 'j' without the number at index 'i - 1' first, no matter true or false**

**2. if we can find a subset to get the remaining sum, '|=' or relational operator means besides directly inherit, we find another choice**

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for(int num : nums) {

sum += num;

}

if(sum % 2 == 1) {

return false;

}

int target = sum / 2;

boolean[][] dp = new boolean[1 + nums.length][1 + target];

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

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

dp[i][0] = true;

}

// For len as 0(no number in nums array pick up),

// given any target, we can not make it

for(int i = 1; i <= target; i++) {

dp[0][i] = false;

}

// process all subsets for all sums

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

for(int j = 1; j <= target; j++) {

// Directly inherit boolean result get the sum 'j' without the

// number at index 'i - 1' first, no matter true or false

dp[i][j] = dp[i - 1][j];

// if we can find a subset to get the remaining sum, '|=' or

// relational operator means besides directly inherit, we find

// another choice

if(j >= nums[i - 1]) {

dp[i][j] |= dp[i - 1][j - nums[i - 1]];

}

}

}

return dp[nums.length][target];

}

}

**Style 3: Initialize boolean[][] dp = new boolean[nums.length][1 + target]**

**1. Directly inherit boolean result get the sum 'j' without the number at index 'i ' first, no matter true or false**

**2. Different than Style 2 since no offset on 'i', since no dummy row added, exclude nums[i] is fine, not nums[i - 1],  if we can find a subset to get the remaining sum, '|=' or relational operator means besides directly inherit, we find another choice**

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for(int num : nums) {

sum += num;

}

if(sum % 2 == 1) {

return false;

}

int target = sum / 2;

boolean[][] dp = new boolean[nums.length][1 + target];

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

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

dp[i][0] = true;

}

// Different than Style 2 since the 1st row not dummy

// with only one number, we can form a subset only when the required sum

// is equal to its value

for(int i = 1; i <= target; i++) {

dp[0][i] = (nums[0] == i ? true : false);

}

// process all subsets for all sums

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

for(int j = 1; j <= target; j++) {

// Direclty inherit boolean result get the sum 'j' without the

// number at index 'i' first, no matter true or false

dp[i][j] = dp[i - 1][j];

// Different than Style 2 since no offset on 'i', since no

// dummy row added, exclude nums[i] is fine, not nums[i - 1]

// if we can find a subset to get the remaining sum, '|=' or

// relational operator means besides directly inherit, we find

// another choice

if(j >= nums[i]) {

dp[i][j] |= dp[i - 1][j - nums[i]];

}

}

}

return dp[nums.length - 1][target];

}

}

**Solution 4: 1D bottom up DP (60 min)**

此时，外层是对数组的循环，内层通过一个 i=num～sum 的循环，求出所有加上当前位置的 num 能到得到 i 的解。外层对数组的循环，每到一个数字 num ，就是指对于子数组[0,...,num]来说，dp[i] 存放着数组[0,...,num]中和为 i 的子数组的个数，那么外层对数组的循环完了之后，dp[i] 存放的就是数组 nums 的和为 i 的子数组的个数，所以，此时 dp[sum] 就是本题的解。

其中，起始位置的dp[0] = true，是整个计算的开端，然后以此类推。

class Solution {

public boolean canPartition(int[] nums) {

int sum = 0;

for(int num : nums) {

sum += num;

}

if(sum % 2 == 1) {

return false;

}

int target = sum / 2;

// dp[j]存放着数组[0, ... , num]中和为j的子数组是否存在

boolean[] dp = new boolean[1 + target];

dp[0] = true;

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

// Must loop backwards

for(int j = target; j >= 0; j--) {

if(j >= nums[i]) {

//dp[j] = dp[j] || dp[j - nums[i]];

dp[j] |= dp[j - nums[i]];

}

}

}

return dp[target];

}

}

**Explanation of why we can degrade 2D to 1D array and why we need loop backwards**

Refer to

https://leetcode.com/problems/partition-equal-subset-sum/discuss/90592/01-knapsack-detailed-explanation

https://leetcode.com/problems/partition-equal-subset-sum/solutions/90592/01-knapsack-detailed-explanation/comments/241664

Yes, the magic is observation from the induction rule/recurrence relation!

For this problem, the induction rule:

If not picking nums[i - 1], then dp[i][j] = dp[i-1][j]

if picking nums[i - 1], then dp[i][j] = dp[i - 1][j - nums[i - 1]]

You can see that if you point them out in the matrix, it will be like:

j

. . . . . . . . . . . .

. . . . . . . . . . . .

. . ? . . ? . . . . . . ?(left): dp[i - 1][j - nums[i], ?(right): dp[i - 1][j]

i . . . . . # . . . . . . # dp[i][j]

. . . . . . . . . . . .

. . . . . . . . . . . .

. . . . . . . . . . . .

. . . . . . . . . . . .

. . . . . . . . . . . .

Optimize to O(2\*n): you can see that dp[i][j] only depends on previous row, so you can

optimize the space by only using 2 rows instead of the matrix. Let's say array1 and array2.

Every time you finish updating array2, array1 have no value, you can copy array2 to array1

as the previous row of the next new row.

Optimize to O(n): you can also see that, the column indices of dp[i - 1][j - nums[i]] and

dp[i - 1][j] are <= j. The conclusion you can get is: the elements of previous row whose

column index is > j(i.e. dp[i - 1][j + 1 : n - 1]) will not affect the update of dp[i][j]

since we will not touch them:

j

. . . . . . . . . . . .

. . . . . . . . . . . .

. . ? . . ? x x x x x x you will not touch x for dp[i][j]

i . . . . . # . . . . . . # dp[i][j]

. . . . . . . . . . . .

. . . . . . . . . . . .

. . . . . . . . . . . .

. . . . . . . . . . . .

. . . . . . . . . . . .

Thus if you merge array1 and array2 to a single array array, if you update array backwards,

all dependencies are not touched!

(n represents new value, i.e. updated)

. . ? . . ? n n n n n n n

#

However if you update forwards, dp[j - nums[i - 1]] is updated already, you cannot use it:

(n represents new value, i.e. updated)

n n n n n ? . . . . . . where another ? goes? Oops, it is overriden, we lost it :(

#

Conclusion:

So the rule is that observe the positions of current element and its dependencies in the matrix.

Mostly if current elements depends on the elements in previous row(most frequent case)/columns,

you can optimize the space.

**Refer to**

<https://leetcode.com/problems/target-sum/solutions/245073/Java-solution-in-Chinese/>

**S1:直接递归**

题目中指出，将数组中的每个数前面加上一个符号（+ 或 -），使得它们构成一个运算式，并且其计算结果等于给定的 S ，求符号的添加种类。

那么也就是说，对于每一个数字，我们可以选择加上它或者减去它，于是，可以直接使用递归，进行2^n次运算，求出所有的符号添加方式，然后再求其中结果为 S 的即可：

public static int findTargetSumWays(int[] nums, int S) {

return ways(nums, S, 0, 0);

}

private static int ways(int[] nums, int S, int pos, int cur) {

if (pos == nums.length) {

return cur == S ? 1 : 0;

} else {

return ways(nums, S, pos+1, cur+nums[pos])

+ ways(nums, S, pos+1, cur-nums[pos]);

}

}

**S2:缓存递归**

上述递归很有可能会增加很多不必要的计算，如对于一个数组[1,1,1,1,1,1]，如果前两个数字的符号分别是[-,+]和[+,-]，那么计算到第三个数字的时候，这两种符号添加方式指向了同一种情况，如果不加处理，肯定要造成后面的[1,1,1,1]的重复计算，此时可以考虑给递归函数加一个缓存。

每个递归函数有两个变量：

1. 当前位置
2. 当前的运算结果

所以初步考虑缓存应该是一个二维数组，此时就要判断这两个变量各自的取值范围：

1. 对于当前的位置，肯定是要在 nums 里面，也就是说，它的范围是 0～n-1
2. 由于数组中所有的数字都是正数，那么必然所有符号取 + 结果最大，所有符号取 - 结果最小，题目里面也指出，所有的数字和是不大于 1000 的，所以，范围取上下 1000 即可，也就是 2001

那么，添加了缓存之后就是：

public static int findTargetSumWays(int[] nums, int S) {

int[][] saved = new int[nums.length][2001];

for (int[] row : saved) {

Arrays.fill(row, -1);

}

return ways2(nums, S, 0, 0, saved);

}

private static int ways2(int[] nums, int S, int pos, int cur, int[][] saved) {

if (pos == nums.length) {

return cur == S ? 1 : 0;

} else {

if (saved[pos][cur+1000] >= 0) return saved[pos][cur+1000];

int ways = ways2(nums, S, pos+1, cur+nums[pos], saved)

+ ways2(nums, S, pos+1, cur-nums[pos], saved);

saved[pos][cur+1000] = ways;

return ways;

}

}

1000 是数组中数字和的上限，如果想要缩减一下内存使用量，也可以直接计算出数字和。

**S3: 动态规划**

动态规划也需要一个数组保存记录值，与递归中使用的缓存数组类似。对于一个数组[1,...,n]来说，如果当前待判断位置是 i ，也就是说已经计算出了前面[1,...,i-1]这部分子数组的结果，每个数字有两种符号添加，也就是说总的应该有2^{i-1}种结果（可能会有重合的），那么在这2^{i-1}种结果加上（或者减去）nums[i] 之后，就会得到2^i种新的结果（当然，有可能在前面的2^{i-1}种结果中，存在 x y ，使得 x-nums[i] == y+nums[i] == cur），此时 cur 对应的解应该是 x y 两个数对应的解的和。那么当整个数组计算完成之后，就可以求出数字 S 对应的解。

public static int findTargetSumWays(int[] nums, int S) {

int sum = 0;

for (int num : nums) sum += num;

if (sum < S || ((S + sum) & 1) == 1) return 0;

int[] dp = new int[(sum<<1) + 1];

dp[nums[0] + sum] = 1;

dp[-nums[0] + sum] += 1;

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

int[] next = new int[(sum<<1) + 1];

for (int j = -sum; j <= sum; j++) {

if (dp[j + sum] > 0) {

next[j + sum + nums[i]] += dp[j + sum];

next[j + sum - nums[i]] += dp[j + sum];

}

}

dp = next;

}

return dp[S + sum];

}

如上，第一个循环是遍历数组，第二个循环则是遍历以求出的2^{i-1}种结果，并将其分别加上（减去）nums[i] 以求出下一个位置的2^i种结果，当然，上述结果肯定是存在重合的，因为所有的运算结果都是在 -sum～sum 的范围内，所以当我们需要遍历已求出的2^{i-1}种结果时，由于会有重合，我们并不知道到底有多少种结果，这时的处理办法就是遍历整个可能的结果（从 -sum 到 sum），当这个结果对应的解不为 0 时，就意味着这是一个2^{i-1}个结果中的一个。

所以，本题中首先给执行了dp[nums[0] + sum] = 1;和dp[-nums[0] + sum] += 1;这两个代码，就是为了先求出2^1的结果，然后之后求第二个位置的2^2的结果的时候，才能利用已求出的结果。

另外，计算下一个位置的结果的时候的代码next[j + sum + nums[i]] += dp[j + sum];中使用的是+=符号，就是将所有导向统一结果的解相加

还有，S 与 sum 是否有什么关系？为什么要判断((S + sum) & 1) == 1？

本题中每个数字前都有一个符号，+ 或 - ，如果将整个数组按照数字之前的符号分成两部分，一部分的符号全是 + ，另一部分的符号全是 - ，然后让两部分的和分别是 x y ，那么有关系式：

x + y = sum

x - y = S

于是有x + x = S + sum = 2 \* x，于是得出结论：S + sum 必然是偶数，所以才有了本题开始的判断。

**S4:划分数组 & 动态规划**

上面的这个关系式S + sum = 2 \* x也可以加以利用，得出(S + sum) / 2 = x，也就是说本题可以变为，使 nums 数组的子数组的和等于 S+sum 的一半，求出符合条件的子数组的个数。

于是有：

public static int findTargetSumWays(int[] nums, int S) {

int sum = 0;

for (int num : nums) sum += num;

if (sum < S || ((sum + S) & 1) == 1) return 0;

sum = (sum + S) >> 1;

int[] dp = new int[sum+1];

dp[0] = 1;

for (int num : nums) {

for (int i = sum; i >= num; i--) {

dp[i] += dp[i-num];

}

}

return dp[sum];

}

此时，外层是对数组的循环，内层通过一个 i=num～sum 的循环，求出所有加上当前位置的 num 能到得到 i 的解。外层对数组的循环，每到一个数字 num ，就是指对于子数组[0,...,num]来说，dp[i] 存放着数组[0,...,num]中和为 i 的子数组的个数，那么外层对数组的循环完了之后，dp[i] 存放的就是数组 nums 的和为 i 的子数组的个数，所以，此时 dp[sum] 就是本题的解。

其中，起始位置的dp[0] = 1，是整个计算的开端，由 0+num=num ，得到dp[num] = 1，然后以此类推。