

## Equal Subset Sum Partition (medium)

We'll cover the following ^

- Problem Statement
- Try it yourself
- Basic Solution
  - Code
  - Time and Space complexity
- Top-down Dynamic Programming with Memoization
  - Code
  - Time and Space complexity
- Bottom-up Dynamic Programming
  - Code
  - Time and Space complexity

### 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 subsets is equal.

#### 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:

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:

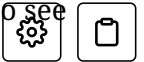
Input: {2, 3, 4, 6}

Output: False

Explanation: The given set cannot be partitioned into two subsets with equal sum.

### Try it yourself #

This problem looks similar to the 0/1 Knapsack problem. Try solving it before moving on to see the solution:



Java	Python3	JS	C++
------	---------	----	-----

```
1 def can_partition(num):
2     # TODO: Write your code here
3     return False
4
5 def main():
6     print("Can partition: " + str(can_partition([1, 2, 3, 4])))
7     print("Can partition: " + str(can_partition([1, 1, 3, 4, 7])))
8     print("Can partition: " + str(can_partition([2, 3, 4, 6])))
9
10
11 main()
12
```

## 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 that  $S$  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  $S/2$ ".

So our brute-force algorithm will look like:

```
1 for each number 'i'
2     create a new set which INCLUDES number 'i' if it does not exceed 'S/2', and recursively
3     process the remaining numbers
4     create a new set WITHOUT number 'i', and recursively process the remaining items
5 return true if any of the above sets has a sum equal to 'S/2', otherwise return false
```

## Code #

Here is the code for the brute-force solution:

Java	Python3	C++	JS
------	---------	-----	----

```
1 def can_partition(num):
2     s = sum(num)
3     # if 's' is a an odd number, we can't have two subsets with equal sum
4     if s % 2 != 0:
5         return False
6
7     return can_partition_recursive(num, s / 2, 0)
8
9
10 def can_partition_recursive(num, sum, currentIndex):
11     # base check
12     if sum == 0:
13         return True
```

```

14
15     n = len(num)
16     if n == 0 or currentIndex >= n:
17         return False
18
19     # recursive call after choosing the number at the `currentIndex`
20     # if the number at `currentIndex` exceeds the sum, we shouldn't process this
21     if num[currentIndex] <= sum:
22         if(can_partition_recursive(num, sum - num[currentIndex], currentIndex + 1)):
23             return True
24
25     # recursive call after excluding the number at the 'currentIndex'
26     return can_partition_recursive(num, sum, currentIndex + 1)
27
28

```



### Time and Space complexity #

The time complexity of the above algorithm is exponential  $O(2^n)$ , where 'n' represents the total number. The space complexity is  $O(n)$ , which will be used to store the recursion stack.

## 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 so we can 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()`.

### Code #

Here is the code for Top-down DP with memoization:

Java

Python3

C++

JS

```

1  def can_partition(num):
2      s = sum(num)
3
4      # if 's' is a an odd number, we can't have two subsets with equal sum
5      if s % 2 != 0:
6          return False
7
8      # initialize the 'dp' array, -1 for default, 1 for true and 0 for false
9      dp = [[-1 for x in range(int(s/2)+1)] for y in range(len(num))]
10     return True if can_partition_recursive(dp, num, int(s / 2), 0) == 1 else False
11
12
13  def can_partition_recursive(dp, num, sum, currentIndex):
14      # base check
15      if sum == 0:
16          return 1

```

```

16     return 1
17
18     n = len(num)
19     if n == 0 or currentIndex >= n:
20         return 0
21
22     # if we have not already processed a similar problem
23     if dp[currentIndex][sum] == -1:
24         # recursive call after choosing the number at the currentIndex
25         # if the number at currentIndex exceeds the sum, we shouldn't process this
26         if num[currentIndex] <= sum:
27             if can_partition_recursive(dp, num, sum - num[currentIndex], currentIndex + 1) == 1:
28                 dp[currentIndex][sum] = 1

```



## Time and Space complexity #

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

## Bottom-up Dynamic Programming #

Let's try to populate our `dp[][]` array from the above solution by 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 the sum 's' from the first 'i' numbers.**

So, for each number at index 'i' ( $0 \leq i < \text{num.length}$ ) and sum 's' ( $0 \leq s \leq S/2$ ), 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:

num\sum	0	1	2	3	4	5
1	T					
{1, 2}	T					
{1,2,3}	T					
{1,2,3,4}	T					

'0' sum can always be found through an empty set

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T					
{1,2,3}	T					
{1,2,3,4}	T					

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

2 of 10

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T				
{1,2,3}	T					
{1,2,3,4}	T					

sum: 1, index:1=> (dp[index-1][sum] , as the 'sum' is less than the number at index '1'

3 of 10

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T			
{1,2,3}	T					
{1,2,3,4}	T					

sum: 2, index:1=> (dp[index-1][sum] || dp[index-1][sum-2])

4 of 10

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T		
{1,2,3}	T					
	T					

{1,2,3,4}

1					
---	--	--	--	--	--



sum: 3, index:1=> (dp[index-1][sum] || dp[index-1][sum-2])

5 of 10

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T					
{1,2,3,4}	T					

sum: 4,5, index:1=> (dp[index-1][sum] || dp[index-1][sum-2])

6 of 10

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T		
{1,2,3,4}	T					

sum: 1,2,3, index:2=> (dp[index-1][sum] || dp[index-1][sum-3])

7 of 10

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T	T	
{1,2,3,4}	T					

sum: 4, index:2=> (dp[index-1][sum] || dp[index-1][sum-3])

8 of 10

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T	T	T
{1,2,3,4}	T					

sum: 5, index:2=> (dp[index-1][sum] || dp[index-1][sum-3])

9 of 10

num\sum	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T	T	T
{1,2,3,4}	T	T	T	T	T	T

sum: 1-5, index:3=> (dp[index-1][sum] || dp[index-1][sum-4])

10 of 10

— []

From the above visualization, we can clearly see that it is possible to partition the given set into two subsets with equal sums, as shown by bottom-right cell:  $dp[3][5] \Rightarrow T$

Code #

Here is the code for our bottom-up dynamic programming approach:

 Java

 Python3

 C++

 JS

```

1 def can_partition(num):
2     s = sum(num)
3
4     # if 's' is a an odd number, we can't have two subsets with same total
5     if s % 2 != 0:
6         return False
7
8     # we are trying to find a subset of given numbers that has a total sum of 's/2'.
9     s = int(s / 2)
10
11     n = len(num)
12     dp = [[False for x in range(s+1)] for y in range(n)]
13
14     # populate the s=0 column, as we can always get 0 sum with an empty set

```

```

14 # populate the s=0 columns, as we can always for 0 sum with an empty set
15 for i in range(0, n):
16     dp[i][0] = True
17
18 # with only one number, we can form a subset only when the required sum is
19 # equal to its value
20 for j in range(1, s+1):
21     dp[0][j] = num[0] == j
22
23 # process all subsets for all sums
24 for i in range(1, n):
25     for j in range(1, s+1):
26         # if we can get the sum 'j' without the number at index 'i'
27         if dp[i - 1][j]:
28             dp[i][j] = dp[i - 1][j]

```



### Time and Space complexity #

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

← Back

Next →

0/1 Knapsack (medium)

Subset Sum (medium)

✓ Completed



Report  
an Issue



Ask a Question

([https://discuss.educative.io/tag/equal-subset-sum-partition-medium\\_\\_pattern--01-knapsack-dynamic-programming\\_\\_grokking-the-coding-interview-patterns-for-coding-questions](https://discuss.educative.io/tag/equal-subset-sum-partition-medium__pattern--01-knapsack-dynamic-programming__grokking-the-coding-interview-patterns-for-coding-questions))