









Equal Subset Sum Partition

We'll cover the following



- Problem Statement
 - Example 1:
 - Example 2:
 - Example 3:
- Try it yourself
- Basic Solution
 - Code
- Top-down Dynamic Programming with Memoization
 - Code
- Bottom-up Dynamic Programming
 - Code

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:

Input: {1, 2, 3, 4}

Output: True

Explanation: The given set can be partitioned into two subsets with equa

1 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 equa

l 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 e

qual sum.

Try it yourself

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



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 number contact the equal subsets must have a sum equal to \$/2. This essentially transforms our problem to: "Find a subset of the given numbers that has a total sum of \$/2".

So our brute-force algorithm will look like:

```
for each number 'i'

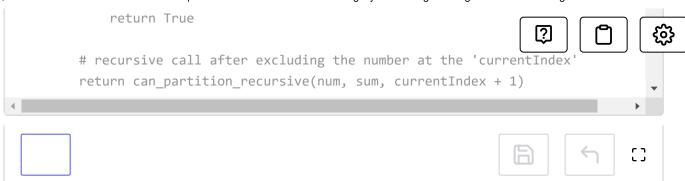
create a new set which INCLUDES number 'i' if it does not exceed 'S/2', an process the remaining numbers

create a new set WITHOUT number 'i', and recursively process the remaining return true if any of the above sets has a sum equal to 'S/2', otherwise ret
```

Code

Here is the code for the brute-force solution:

```
Python3
    def can_partition(num):
      s = sum(num)
      # if 's' is a an odd number, we can't have two subsets with equal sum
      if s % 2 != 0:
        return False
      return can_partition_recursive(num, s / 2, 0)
    def can partition recursive(num, sum, currentIndex):
      # base check
      if sum == 0:
        return True
      n = len(num)
      if n == 0 or currentIndex >= n:
        return False
      # recursive call after choosing the number at the `currentIndex`
      # if the number at `currentIndex` exceeds the sum, we shouldn't process th
      if num[currentIndex] <= sum:</pre>
        if(can_partition_recursive(num, sum - num[currentIndex], currentIndex +
```



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

Top-down Dynamic Programming with Memoization#

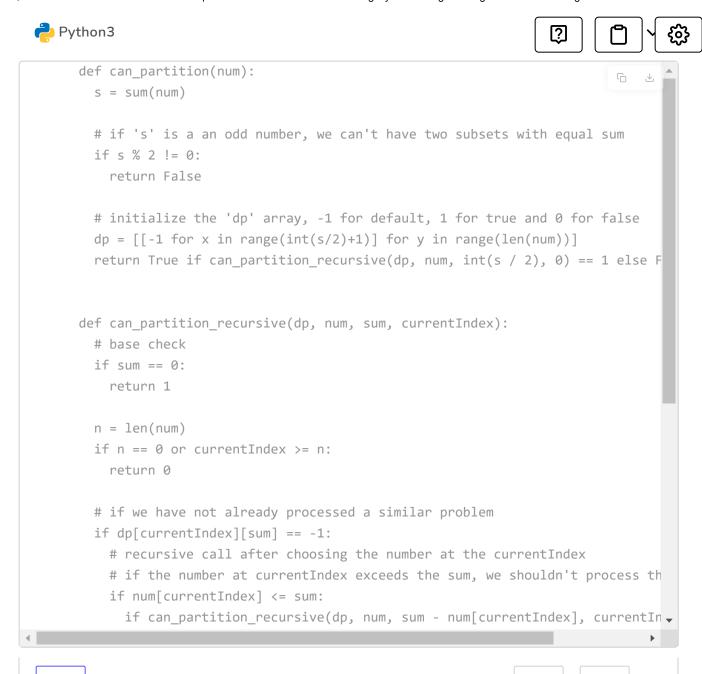
We can use memoization to overcome the overlapping sub-problems. As stated in previous lessons, memoization is when we store the results of all the previously solved sub-problems return the results from memory if we encounter a problem that has already been solved.

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 <code>canPartitionRecursive()</code>.

Code#

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





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.

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' [?] sum 's' from the first 'i' numbers.

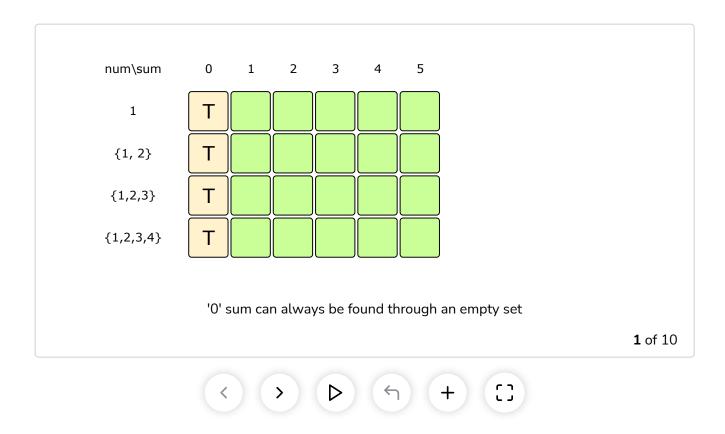


So, for each number at index 'i' (0 \le i \le num.length) and sum 's' (0 \le s \le S/2), we have two options:

- 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:



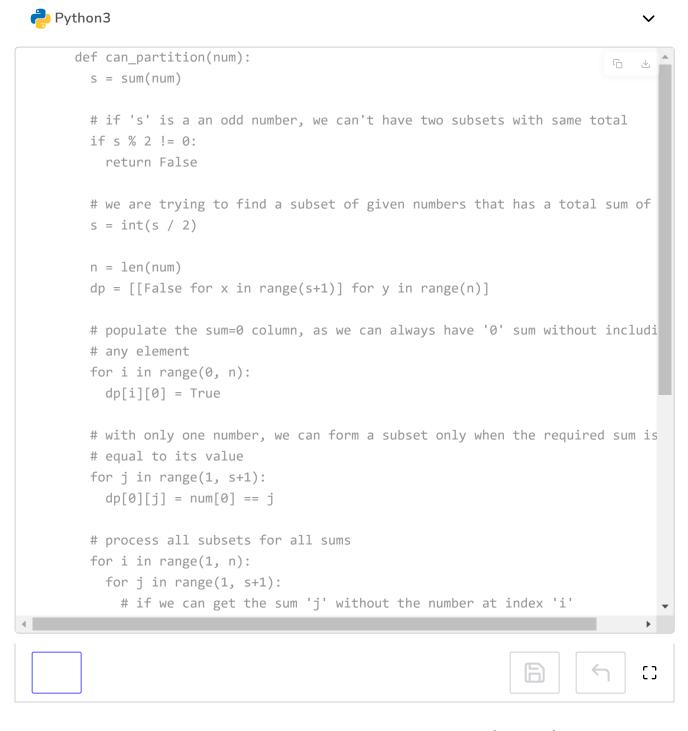
From the above visualization, we can clearly see that it is possible to partition the given set into two subsets with equal sums, as shown by bottom-right cell: $dp[3][5] \Rightarrow T$



Code #



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



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.



