

# Minimum Subset Sum Difference (hard)

#### We'll cover the following

^

• Problem Statement

<del>\*</del>

- Example 1:
- Example 2:
- Example 3:
- Basic Solution
  - Code
  - Time and Space complexity
- Top-down Dynamic Programming with Memoization
  - Code
- Bottom-up Dynamic Programming
  - Code
  - Time and Space complexity

#### Problem Statement #

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

#### Example 1:#

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

## Example 2: #

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

#### Example 3: #

```
Input: {1, 3, 100, 4}
Output: 92
Explanation: We can partition the given set into two subsets where minimum absolut
```

e difference between the sum of numbers is '92'. Here are the two subsets: {1, 3, 4} & {100}

### **Basic Solution #**

This problem follows the **0/1 Knapsack pattern** and can be converted into a Subset Sum (https://www.educative.io/collection/page/5668639101419520/5671464854355968/612696812473 5488/) problem.

Let's assume S1 and S2 are the two desired subsets. A basic brute-force solution could be to try adding each element either in S1 or S2 in order to find the combination that gives the minimum sum difference between the two sets.

So our brute-force algorithm will look like:

```
1 for each number 'i'
2 add number 'i' to S1 and recursively process the remaining numbers
3 add number 'i' to S2 and recursively process the remaining numbers
4 return the minimum absolute difference of the above two sets
```

Code #

Here is the code for the brute-force solution:

```
👙 Java
           Python3
                         G C++
                                     JS JS
 1 def can_partition(num):
 2
      return can_partition_recursive(num, 0, 0, 0)
 3
 4
 5
    def can_partition_recursive(num, currentIndex, sum1, sum2):
 6
      # base check
 7
      if currentIndex == len(num):
 8
        return abs(sum1 - sum2)
 9
10
      # recursive call after including the number at the currentIndex in the first set
11
      diff1 = can partition recursive(
12
        num, currentIndex + 1, sum1 + num[currentIndex], sum2)
13
      # recursive call after including the number at the currentIndex in the second set
14
15
      diff2 = can_partition_recursive(
        num, currentIndex + 1, sum1, sum2 + num[currentIndex])
16
17
18
      return min(diff1, diff2)
19
20
21 def main():
22
      print("Can partition: " + str(can_partition([1, 2, 3, 9])))
      print("Can partition: " + str(can_partition([1, 2, 7, 1, 5])))
23
24
      print("Can partition: " + str(can_partition([1, 3, 100, 4])))
25
26
27 main()
28
                                                                                        \leftarrow
>
```



Because of the two recursive calls, 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 is used to store the recursion stack.

# 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 #

Here is the code:

```
👙 Java
            Python3
                          ⊘ C++
                                       JS JS
 1 def can_partition(num):
      s = sum(num)
 3
       dp = [[-1 \text{ for } x \text{ in } range(s+1)] \text{ for } y \text{ in } range(len(num))]
 4
       return can_partition_recursive(dp, num, 0, 0, 0)
 5
 6
 7 def can_partition_recursive(dp, num, currentIndex, sum1, sum2):
 8
      # base check
 9
      if currentIndex == len(num):
10
         return abs(sum1 - sum2)
11
12
      # check if we have not already processed similar problem
13
      if dp[currentIndex][sum1] == -1:
         # recursive call after including the number at the currentIndex in the first set
14
15
         diff1 = can_partition_recursive(
16
           dp, num, currentIndex + 1, sum1 + num[currentIndex], sum2)
17
         # recursive call after including the number at the currentIndex in the second set
18
19
         diff2 = can_partition_recursive(
20
           dp, num, currentIndex + 1, sum1, sum2 + num[currentIndex])
21
22
         dp[currentIndex][sum1] = min(diff1, diff2)
23
24
       return dp[currentIndex][sum1]
25
26
27 def main():
28
       print("Can partition: " + str(can partition([1, 2, 3, 9])))
\triangleright
                                                                                                 []
```

# **Bottom-up Dynamic Programming #**

Let's assume 'S' represents the total sum of all the numbers. So, in this problem, we are trying 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

5488/), 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.

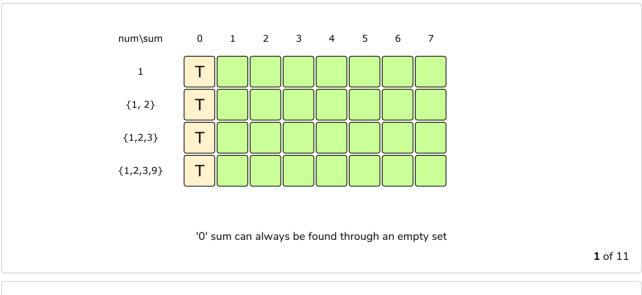
Essentially, we need to calculate all the possible sums up to 'S/2' for all numbers. So how can we populate the array db[TotalNumbers][S/2+1] in the bottom-up fashion?

For every possible sum 's' (where  $0 \le s \le S/2$ ), 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]]

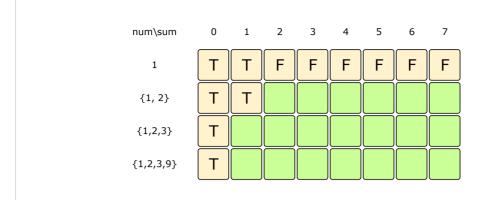
If either of the two above scenarios is true, we can find a subset with a sum equal to 's'. We should dig into this before we can learn how to find the closest subset.

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



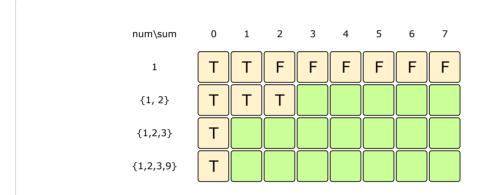
num\sum	0	1	2	3	4	5	6	7
1	Т	T	F	F	F	F	F	F
{1, 2}	Т							
{1,2,3}	T							
{1,2,3,9}	T							
With only one nu	ımber, we	can for	m a su	ıbset oı	nly wh	en the	require	ed sum





sum: 1, index:1=> (dp[index-1][sum] , as the 'sum' is less than the number at index '1' (i.e., 1 < 2)

**3** of 11



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

**4** of 11

num\sum	0	1	2	3	4	5	6	7
1	Т	Т	F	F	F	F	F	F
{1, 2}	T	T	T	T				
{1,2,3}	T							
{1,2,3,9}	T							

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

**5** of 11

num\sum	0	1	2	3	4	5	6	7
1	T	Т	F	F	F	F	F	F
{1, 2}	T	T	Т	T	F	F	F	F
{1,2,3}	T							
	_							



sum: 5,6, index:2=> (dp[index-1][sum] || dp[index-1][sum-3])

**9** of 11





- []

The above visualization tells us that it is not possible to find a subset whose sum is equal to '7'. So what is the closest subset we can find? We can find the subset if we start moving backwards in the last row from the bottom right corner to find the first 'T'. The first "T" in the diagram above is the sum '6', which means that we can find a subset whose sum is equal to '6'. This means the other set will have a sum of '9' and the minimum difference will be '3'.

#### Code #

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

```
Python3
🍨 Java
                         ○ C++
                                     JS JS
22
            dp[i][j] = dp[i - 1][j - num[i]]
23
24
25
      # find the largest index in the last row which is true
26
      for i in range(int(s/2), -1, -1):
27
        if dp[n - 1][i]:
          sum1 = i
28
29
          break
30
```

```
31
      sum2 = s - sum1
32
      return abs(sum2 - sum1)
33
34
35
    def main():
36
      print("Can partition: " + str(can_partition([1, 2, 3, 9])))
      print("Can partition: " + str(can_partition([1, 2, 7, 1, 5])))
37
      print("Can partition: " + str(can_partition([1, 3, 100, 4])))
38
39
40
41
    main()
42
43
44
45
46
47
48
                                                                                           ←
\triangleright
```

## Time and Space complexity #

The above solution 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.

