SET PARTITION PROBLEM

Problem Statement: Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is same.

Example:

arr[] = {1, 5, 11, 5}

Output: true

The array can be partitioned as {1, 5, 5} and {11}

arr[] = {1, 5, 3}

Output: false

The array cannot be partitioned into equal sum sets.

Solution:

Following are the two main steps to solve this problem:
Calculate sum of the array.

1) If sum is odd, there cannot be two subsets with equal sum, so return false.
2) If sum of array elements is even, calculate sum/2 and find a subset of array with sum equal to sum/2.

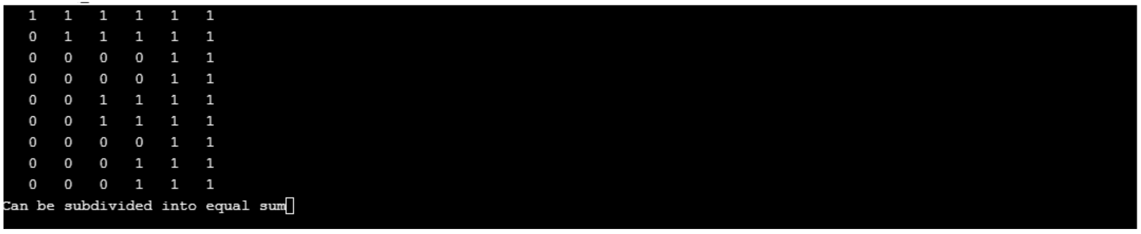
The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array partition[][] of size (sum/2)*(n+1). And we can construct the solution in bottom up manner such that every filled entry has following property:

partition[i][j] = true, if a subset of {array[0], array[1], ..array[j-1]} has sum equal to i, otherwise false.

Output:

Values in partition table is as follows:



```
   1   1   1   1   1   1
   0   1   1   1   1   1
   0   0   0   0   1   1
   0   0   0   0   1   1
   0   0   1   1   1   1
   0   0   1   1   1   1
   0   0   0   0   1   1
   0   0   0   1   1   1
   0   0   0   1   1   1
Can be subdivided into equal sum
```

Time Complexity: O(sum*n)
Auxiliary Space: O(sum*n)

Algorithm:

If sum of all the elements in the given set is an odd number say '2n+1', then the best we might be able to do is to partition the given set into two subsets - one with sum 'n' and another with sum 'n+1'. In other words, if sum of all the elements in given set is odd, there is no way the set can be partitioned into two subsets with equal sum and therefore we return 'false' in this case.

Now if the sum of all the elements in the given set is an even number say '2n', then all we need to do is to find out a subset say 'S1' of the given set such that sum of elements in that subset is 'n'. The remaining subset(obtained by excluding elements in 'S1') then is guaranteed to have sum = 'n'. For example, the sum of all elements for this set {15,5,15,20,45} is 100. And the subset {5,45} has sum = 50. Now the remaining subset obtained by excluding elements 5 and 45 is {15,15,20} which also has sum = 50.

Notice how we have reduced the original problem into the problem of finding a subset with sum = 'n' when sum of all elements in given set is '2n'. Let's try to solve this modified problem now.

I am using dynamic programming(tabulation) approach to solve this problem. We will fill up the 'solution' array shown below

1	1	1	1	1	1
0	1	1	1	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	1	1	1	1
0	0	1	1	1	1
0	0	0	0	1	1
0	0	0	1	1	1
0	0	0	1	1	1

Here, value of solution[i][j] represents if sum of 'i' could be obtained by any subset of the set having first 'j' elements (from the given set) in it. For example, for the set {7,5,3,5} solution[8][3] would be 'true' since sum of 8 could be obtained by a subset({5,3}) of the set {7,5,3}(set obtained by considering first 3 elements of the given set).

Here is how we fill up this solution array -
Initialization:
1. All elements in 0th row are initialized to 'true' because for any given set, we can always find

an empty subset with sum = '0'

2. All elements in 0th column except the element in the 0th row are initialized to 'false' since no sum except 0 could be obtained by any subset of the empty set({}).

Rules for filling up solution[i][j]:
1. If solution[i][j-1] is 'true' then solution[i][j] is also 'true'. This is because if there exists a subset(with sum = 'i') of set formed by first 'j-1' elements(of given set) then that same subset will also be a subset for the set formed by first 'j' elements.

2. If solution[i][j-1] is 'true' then solution[(i + set[j-1])][j] is also 'true'. This is because if there exists a subset(with sum = 'i') of set formed by first 'j-1' elements(of given set) then inserting 'j'th element into that subset results in a new subset which will have sum = 'i' + value of 'j'th element. This sum is represented by the row = 'i + set[j-1]'. Remember indexing scheme used here is 0 based and therefore value of 'j'th element is represented by set[j-1].

3. If solution[i][j] is not set to 'true' using above two rules then set solution[i][j] to 'false'. Because there is no other case other than above two cases in which solution[i][j] would be 'true'.

Using above two rules and initialization, function 'partition(int requiredSum, int[] set)' fills up 'solution' array as shown below.

1	1	1	1	1	1
0	1	1	1	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	1	1	1	1
0	0	1	1	1	1
0	0	0	0	1	1
0	0	0	1	1	1
0	0	0	1	1	1

Return value: Note that if we are looking for a subset of the given set with sum = 'requiredSum' then number of rows(numRows) for above 'solution' array would be 'requiredSum + 1' and number of columns(numCols) would be 'size of given set + 1'. Once we fill up this 'solution' array, we need to return the value of solution[numRows-1][numCols-1].

References:

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman,*The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).Google Scholar
2. R. Garfinkel and G. Nemhauser,*The set partitioning problem: Set covering problem with equality constraints*, Operations Research 17, no. 5 (1969), pp. 848–856.Google Scholar
3. E. Horowitz and S. Sahni,*Fundamentals of Computer Algorithms*, Computer Science Press (1978).Google Scholar
4. C. H. Lin,*Corporate Tax Structures and a Special Class of Set Partitioning Problems*, Ph. D. Thesis, Department of Operation Research, Case Western Reserve University, Cleveland, OH (1976).Google Scholar
5. C. H. Lin and H. M. Salkin,*Aggregation of subsidiary firms for minimal unemployment compensation payments via integer programming*, Management Science 25, no. 4 (1979), pp. 405–408.Google Scholar