Subset Sum Problem

Problem Link - https://practice.geeksforgeeks.org/problems/subset-sum-problem-1611555638/1

Problem statement - Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Output: True
There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
There is no subset that adds up to 30.

----------Identification----------

Question aries that how can we say that this Subset sum problem is Like O-1 knapsack ?

Knapsack	Subset Sum
1. Item array is given	1. Array is given
2. And also the capacity of the knapsack	2. And sum is given
3. And for every item there is choice	3. And here also every item have two choice

So, you surely say that Subset sum is one of the variations of 0-1 knapsack problem.
The same approach is used to find the result but in a slightly different way.

----------Recursive solution----------

1. **Condition.**
   For every item in the array we have two choices: either we take the item or not.
2. **Condition no 2**
   And we check every item's value if the value is greater or smaller than the given sum then do some stuff.

Let see how an you find the base condition
Start from the end of the array,
So for finding the base condition you have to choose the smallest valid input for this array.
Let's suppose
Example no 1.
Int n = arr.size();
Given sum = 0 and n = 0 means empty array

Can you make a subset with given values , yes sure the empty subset in one of the answers so we return true for that case.

Example no 2.

Int n = arr.size();

Given sum = 0 and n = 2 means array contain some element [5 , 6]

Can you make a subset with given values , yes sure the empty subset in one of the answers so we return true for that case.

Example no 3.

Int n = arr.size();

Given sum = 5 and n = 0 means empty array

Can you make a subset with given values , NO because we can't make a subset without any element. In example no 1 , the sum is zero so an empty subset is one of the answers.

Base condition

if(sum == 0) return true;
if(n == 0) return false;

Now according to our conditions

Firstly we check for array value is less than or equal to given sum

If element value is less than or equal to given sum we have two choices: include the element in the subset or not.

Else we have only one option that is not included in our subset.

```
class Solution{
public:
  bool solve(vector<int>arr , int sum , int n ){
      if(sum == 0) return true;
      if(n == 0) return false;
      if(arr[n-1] <= sum){
          return solve(arr , sum - arr[n-1] , n-1) || solve(arr , sum ,
n-1);

      }else{
          return solve(arr , sum, n-1);
      }
  }
    bool isSubsetSum(vector<int>arr, int sum){
        // code here
        int n = arr.size();
        return solve(arr , sum , n);
    }
};
```

So here we can say that this code is similar to knapsack

1. Change in base condition according to our question
2. Here we are dealing with true and false means booleans value so we can't use max term that doesn't make any sense in boolean,
3. So max is replace with "or" , "||"
4. And rest is similar to knapsack

Time complexity for this recursive solution
Complexity Analysis: The above solution may try all subsets of a given set in the worst case. Therefore the time complexity of the above solution is exponential. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).

----------Dp solution Memoization----------

Method

1. In this method, we also follow the recursive approach but In this method, we use another 2-D matrix when we first initialize with -1 or any negative value.
2. In this method, we avoid the few recursive calls which repeat themselves, that's why we use a 2-D matrix. In this matrix we store the value of the previous call value.

Change few line in recursive code to make memoization

1. Initialize the matrix and set it to -1 `vector<vector<int>> dp(n+1 , vector<int>(sum+1 , -1));`
2. Now check if current point is already present in the matrix or not
   For that `if(dp[n][sum] != -1)`
   If present then return that value.
3. Before every recursive call store that value in the matrix .
4. And lastly in the main function convert it into boolean.

Complexity Analysis:

Time Complexity: O(sum*n), where sum is the 'target sum' and 'n' is the size of the array.

Auxiliary Space: O(sum*n) + O(n) -> O(sum*n) = the size of 2-D array is sum*n and O(n)=auxiliary stack space.

```
class Solution{
public:
  bool solve(vector<int>arr , int sum , int n , vector<vector<int>>&dp ){
      if(sum == 0) return 1;
      if(n == 0) return 0;
      if(dp[n][sum] != -1) return dp[n][sum];
```

```
      if(arr[n-1] <= sum){
          return dp[n][sum] =  solve(arr , sum - arr[n-1] , n-1 , dp) ||
solve(arr , sum , n-1 , dp);

      }else{
          return dp[n][sum] = solve(arr , sum, n-1 , dp);
      }
  }
    bool isSubsetSum(vector<int>arr, int sum){
        // code here

         int n = arr.size();
        vector<vector<int>> dp(n+1 , vector<int>(sum+1 , -1));

        if(solve(arr , sum , n , dp)) return true;
        else return false;
    }
};
```

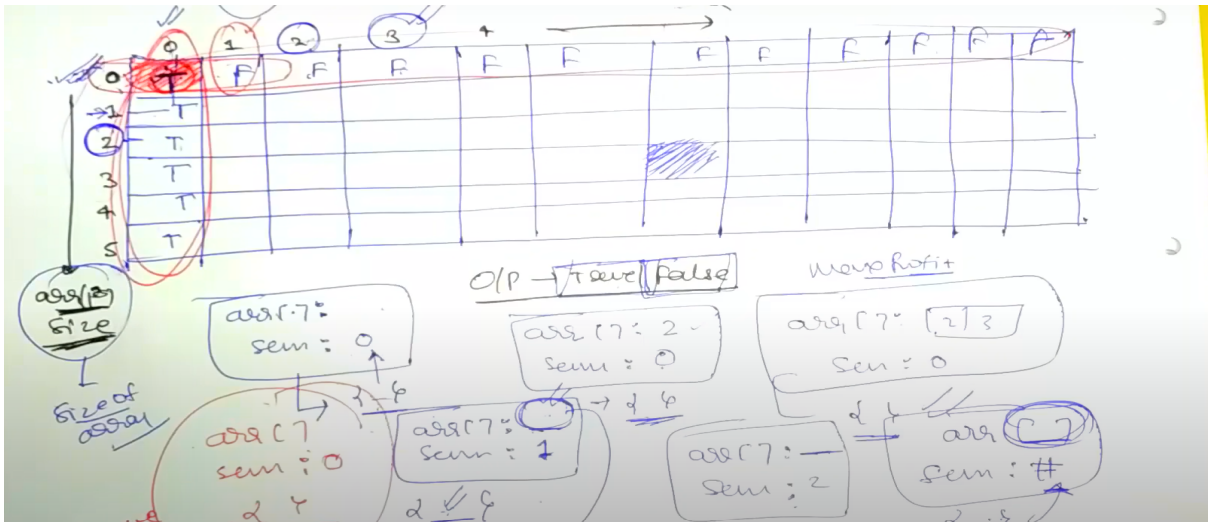
----------Top Down Approach DP----------

Changes made to convert recursive code to top down.

1. So, firstly we will create a 2d matrix to store the value of size dp[n+1][sum+1].
2. In initialization step

   Let's suppose

For the above three examples we conclude that the first row fill with TRUE and first column fill with FALSE and dp[0][0] fill with TRUE.



// Code for initialization step

Firstly we created a 2d matrix and filled false in all the rows and columns

```
vector<vector<bool>> dp(n+1 , vector<bool>(sum+1 , false));
```

And now the first column is filled with “true” because if the sum = 0, then there is a possibility that an empty subset is the answer.

```
for(int i =0 , j=0; i<n+1; i++){
            dp[i][j] = true;
    }
```

And we try to fill the rest of the matrix according to previous found values.

```
class Solution{
public:

    bool isSubsetSum(vector<int>arr, int sum){
        // code here

         int n = arr.size();
        vector<vector<bool>> dp(n+1 , vector<bool>(sum+1 , false));
        for(int i =0 , j=0; i<n+1; i++){
            dp[i][j] = true;
        }
         for(int i =1; i<n+1; i++){
            for(int j = 1; j<sum+1; j++){
               if(arr[i-1] <= j){
                   dp[i][j] = (dp[i-1][j - arr[i-1]] || dp[i-1][j] );
               }else{
                   dp[i][j] = dp[i-1][j];
               }
            }
        }

       return dp[n][sum];
    }
};
```

Complexity Analysis:

Time Complexity: O(sum*n), where sum is the ‘target sum’ and ‘n’ is the size of the array.

Auxiliary Space: O(sum*n), as the size of 2-D array is sum*n. + O(n) for recursive stack space