Subset Sum Problem

Given a set of non-negative integers and a value **sum**, the task is to check if there is a subset of the given set whose sum is equal to the given **sum**.

Examples:

Input: *set[] = {3, 34, 4, 12, 5, 2}, sum = 9*

Output: *True*

Explanation: *There is a subset (4, 5) with sum 9.*

Input: *set[] = {3, 34, 4, 12, 5, 2}, sum = 30*

Output: *False*

Explanation: *There is no subset that add up to 30.*

Subset Sum Problem using Recursion:

For the recursive approach, there will be two cases.

- *Consider the 'last' element to be a part of the subset. Now the **new required sum = required sum – value of 'last' element**.*
- *Don't include the 'last' element in the subset. Then the **new required sum = old required sum**.*

In both cases, the number of available elements decreases by 1.

Mathematically the recurrence relation will look like the following:

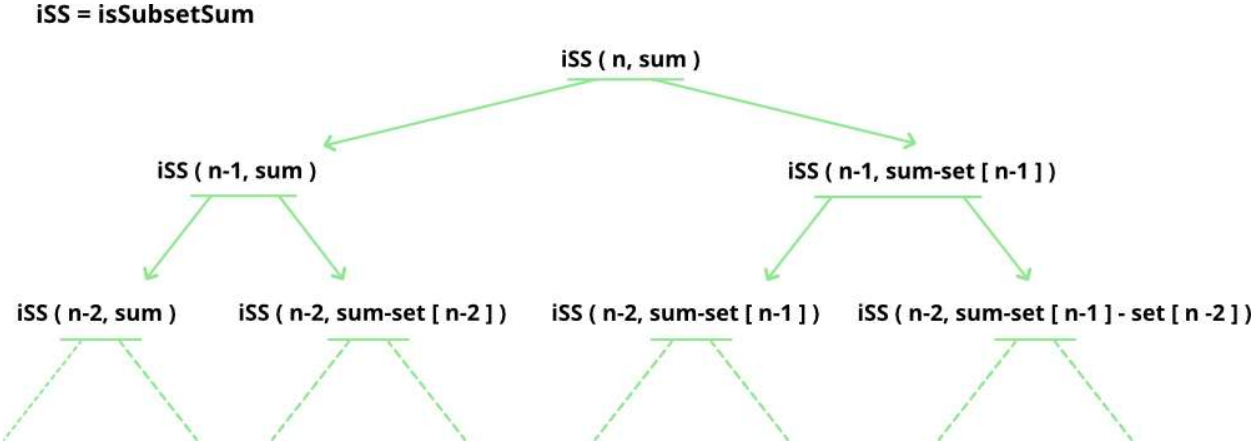
isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) | isSubsetSum(set, n-1, sum-set[n-1])

Base Cases:

isSubsetSum(set, n, sum) = false, if sum > 0 and n = 0

isSubsetSum(set, n, sum) = true, if sum = 0

The structure of the recursion tree will be like the following:



Structure of the recursion tree of the above recursion formula

Follow the below steps to implement the recursion:

- Build a recursive function and pass the index to be considered (here gradually moving from the last end) and the remaining sum amount.
- For each index check the base cases and utilize the above recursive call.
- If the answer is true for any recursion call, then there exists such a subset. Otherwise, no such subset exists.

Below is the implementation of the above approach.

C++

```
// A recursive solution for subset sum problem


#include <bits/stdc++.h>
using namespace std;


// Returns true if there is a subset
// of set[] with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0)
        return false;


    // If last element is greater than sum,
    // then ignore it
    if (set[n - 1] > sum)
        return isSubsetSum(set, n - 1, sum);
```

```
    // Else, check if sum can be obtained by any
    // of the following:
    // (a) including the last element
    // (b) excluding the last element
    return isSubsetSum(set, n - 1, sum)
           || isSubsetSum(set, n - 1, sum - set[n - 1]);
}

// Driver code
int main()
{
    int set[] = { 3, 34, 4, 12, 5, 2 };
    int sum = 9;
    int n = sizeof(set) / sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        cout << "Found a subset with given sum";
    else
        cout << "No subset with given sum";
    return 0;
}
```

Output

```
Found a subset with given sum
```

Complexity Analysis:

- **Time Complexity:** O(2^n) The above solution may try all subsets of the given set in worst case. Therefore 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).
- **Auxiliary Space:** O(n) where n is recursion stack space.

Subset Sum Problem using **Memoization**:

As seen in the previous recursion method, each state of the solution can be uniquely identified using two variables – the index and the remaining sum. So create a 2D array to store the value of each state to avoid recalculation of the same state.

Below is the implementation of the above approach:

C++

```
// CPP program for the above approach
#include <bits/stdc++.h>
using namespace std;

// Taking the matrix as globally
int tab[2000][2000];

// Check if possible subset with
// given sum is possible or not
int subsetSum(int a[], int n, int sum)
{
    // If the sum is zero it means
    // we got our expected sum
    if (sum == 0)
        return 1;
```

```
    if (n <= 0)
        return 0;

    // If the value is not -1 it means it
    // already call the function
    // with the same value.
    // it will save our from the repetition.
    if (tab[n - 1][sum] != -1)
        return tab[n - 1][sum];

    // If the value of a[n-1] is
    // greater than the sum.
    // we call for the next value
    if (a[n - 1] > sum)
        return tab[n - 1][sum] = subsetSum(a, n - 1, sum);
    else
    {
        // Here we do two calls because we
        // don't know which value is
        // full-fill our criteria
        // that's why we doing two calls
        return tab[n - 1][sum] = subsetSum(a, n - 1, sum) ||
                        subsetSum(a, n - 1, sum - a[n - 1]);
    }
}

// Driver Code
```

```
int main()
{
    // Storing the value -1 to the matrix
    memset(tab, -1, sizeof(tab));
    int n = 5;
    int a[] = {1, 5, 3, 7, 4};
    int sum = 12;

    if (subsetSum(a, n, sum))
    {
        cout << "YES" << endl;
    }
    else
        cout << "NO" << endl;

}
```

Output
YES

Complexity Analysis:

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

Subset Sum Problem using **Dynamic Programming**:

To solve the problem in Pseudo-polynomial time we can use the Dynamic programming approach.

*So we will create a 2D array of size (n + 1) * (sum + 1) of type **boolean**. The state **dp[i][j]** will be **true** if there exists a subset of elements from **set[0 . . . i]** with **sum value = 'j'.***

The dynamic programming relation is as follows:

if (A[i-1] > j)

dp[i][j] = dp[i-1][j]

else

dp[i][j] = dp[i-1][j] OR dp[i-1][j-set[i-1]]

This means that if the current element has a value greater than the 'current sum value' we will copy the answer for previous cases and if the current sum value is greater than the 'ith' element we will see if any of the previous states have already experienced the **sum= j OR any previous states experienced a value 'j – set[i]'** which will solve our purpose.

Illustration:

See the below illustration for a better understanding:

*Consider **set[] = {3, 4, 5, 2}** and **sum = 6**.*

The table will look like the following where the column number represents the sum and the row number represents the index of set[]:

	0	1	2	3	4	5	6
no element (0)	T	F	F	F	F	F	F
0 (3)	T	F	F	T	F	F	F
1 (4)	T	F	F	T	T	F	F
2 (5)	T	F	F	T	T	T	F
3 (2)	T	F	T	T	T	T	T

Below is the implementation of the above approach:

C++

```cpp
// A Dynamic Programming solution
// for subset sum problem
#include <bits/stdc++.h>
using namespace std;

// Returns true if there is a subset of set[]
// with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // The value of subset[i][j] will be true if
    // there is a subset of set[0..j-1] with sum
    // equal to i
    bool subset[n + 1][sum + 1];

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        subset[i][0] = true;

    // If sum is not 0 and set is empty,
    // then answer is false
    for (int i = 1; i <= sum; i++)
        subset[0][i] = false;

    // Fill the subset table in bottom up manner
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (j < set[i - 1])
```

```
                subset[i][j] = subset[i - 1][j];
            if (j >= set[i - 1])
                subset[i][j]
                    = subset[i - 1][j]
                      || subset[i - 1][j - set[i - 1]];
        }
    }

    return subset[n][sum];
}

// Driver code
int main()
{
    int set[] = { 3, 34, 4, 12, 5, 2 };
    int sum = 9;
    int n = sizeof(set) / sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        cout << "Found a subset with given sum";
    else
        cout << "No subset with given sum";
    return 0;
}
```

Output

```
Found a subset with given sum
```

Complexity Analysis:

- **Time Complexity:** O(sum * n), where **n** is the size of the array.
- **Auxiliary Space:** O(sum*n), as the size of the 2-D array is **sum*n**.

Subset Sum Problem using **Dynamic Programming** with space optimization to linear:

Idea:

In previous approach of dynamic programming we have derive the relation between states as given below:

if (A[i-1] > j)

dp[i][j] = dp[i-1][j]

else

dp[i][j] = dp[i-1][j] OR dp[i-1][j-set[i-1]]

If we observe that for calculating current dp[i][j] state we only need previous row dp[i-1][j] or dp[i-1][j-set[i-1]].

There is no need to store all the previous states just one previous state is used to compute result.

Approach:

- Define two arrays **prev** and **curr** of size **Sum+1** to store the just previous row result and current row result respectively.
- Once **curr** array is calculated then **curr** becomes our **prev** for the next row.
- When all rows are processed the answer is stored in **prev** array.

Implementation of the above approach:

C++

```
// A Dynamic Programming solution
// for subset sum problem with space optimization
#include <bits/stdc++.h>
using namespace std;
```

```
// Returns true if there is a subset of set[]
// with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{

    vector<bool> prev(sum + 1);

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        prev[0] = true;

    // If sum is not 0 and set is empty,
    // then answer is false
    for (int i = 1; i <= sum; i++)
        prev[i] = false;

    // curr array to store the current row result generated
    // with help of prev array
    vector<bool> curr(sum + 1);

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (j < set[i - 1])
                curr[j] = prev[j];
            if (j >= set[i - 1])
                curr[j] = prev[j] || prev[j - set[i - 1]];
        }
```

```
        // now curr becomes prev for i+1 th element
        prev = curr;
    }

    return prev[sum];
}

// Driver code
int main()
{
    int set[] = { 3, 34, 4, 12, 5, 2 };
    int sum = 9;
    int n = sizeof(set) / sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        cout << "Found a subset with given sum";
    else
        cout << "No subset with given sum";
    return 0;
}
```

Output

```
Found a subset with given sum
```

Complexity Analysis:

- **Time Complexity:** O(sum * n), where **n** is the size of the array.
- **Auxiliary Space:** O(sum), as the size of the 1-D array is **sum+1**.