Subset Sum Problem – Dynamic Programming Solution

Given a set of positive integers and an integer `k`, check if there is any non-empty subset that sums to `k`.

For example,

```
Input:

A = { 7, 3, 2, 5, 8 }
k = 14

Output: Subset with the given sum exists

Subset { 7, 2, 5 } sums to 14
```

Practice this problem

A naive solution would be to cycle through all subsets of `n` numbers and, for every one of them, check if the subset sums to the right number. The running time is of order $O(2^n.n)$ since there are 2^n subsets, and to check each subset, we need to sum at most `n` elements.

A better exponential-time algorithm uses recursion. Subset sum can also be thought of as a special case of the 0–1 Knapsack problem. For each item, there are two possibilities:

1. Include the current item in the subset and recur for the remaining items with the remaining total.
2. Exclude the current item from the subset and recur for the remaining items.

Finally, return true if we get a subset by including or excluding the current item; otherwise, return false. The recursion's base case would be when no items are left, or the sum becomes negative. Return true when the sum becomes 0, i.e., the subset is found.

Following is the C++, Java, and Python implementation of the idea:

C++

```
#include <iostream>
#include <vector>
using namespace std;

// Returns true if there exists a subsequence of `A[0…n]` with the giv
bool subsetSum(vector<int> const &A, int n, int k)
{
    // return true if the sum becomes 0 (subset found)
    if (k == 0) {
        return true;
    }

    // base case: no items left, or sum becomes negative
    if (n < 0 || k < 0) {
        return false;
    }

    // Case 1. Include the current item `A[n]` in the subset and recur
    // for the remaining items `n-1` with the remaining total `k-A[n]`
    bool include = subsetSum(A, n - 1, k - A[n]);

    // Case 2. Exclude the current item `A[n]` from the subset and rec
    // the remaining items `n-1`
    bool exclude = subsetSum(A, n - 1, k);

    // return true if we can get subset by including or excluding the
    // current item
    return include || exclude;
}
```

```
32  int main()
33  {
34      // Input: a set of items and a sum
35      vector<int> A = { 7, 3, 2, 5, 8 };
36      int k = 14;
37
38      // total number of items
39      int n = A.size();
40
41      if (subsetSum(A, n - 1, k)) {
42          cout << "Subsequence with the given sum exists";
43      }
44      else {
45          cout << "Subsequence with the given sum does not exist";
46      }
47
48      return 0;
49  }
```

Download Run Code

Output:

```
Subsequence with the given sum exists
```

Java

Python

The time complexity of the above solution is exponential and occupies space in the call stack.

The problem has an optimal substructure. That means the problem can be broken down into smaller, simple “subproblems”, which can further be divided into yet simpler, smaller subproblems until the solution becomes trivial. The above solution also exhibits overlapping subproblems. If we draw the solution’s recursion tree, we can see that the same subproblems are getting computed repeatedly.

We know that problems with optimal substructure and overlapping subproblems can be solved using dynamic programming, where subproblem solutions are *memo*ized rather than computed again and again. Following is the *memo*ized implementation in C++, Java, and Python, which

C++

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

// Returns true if there exists a subsequence of `A[0…n]` with the giv
bool subsetSum(vector<int> const &A, int n, int k, auto &lookup)
{
    // return true if the sum becomes 0 (subset found)
    if (k == 0) {
        return true;
    }

    // base case: no items left, or sum becomes negative
    if (n < 0 || k < 0) {
        return false;
    }

    // construct a unique map key from dynamic elements of the input
    string key = to_string(n) + "|" + to_string(k);

    // if the subproblem is seen for the first time, solve it and
    // store its result in a map
    if (lookup.find(key) == lookup.end())
    {
        // Case 1. Include the current item `A[n]` in the subset and r
        // for the remaining items `n-1` with the remaining total `k-A
        bool include = subsetSum(A, n - 1, k - A[n], lookup);

        // Case 2. Exclude the current item `A[n]` from the subset and
        // the remaining items `n-1`
        bool exclude = subsetSum(A, n - 1, k, lookup);

        // assign true if we can get subset by including or excluding
        // current item
        lookup[key] = include || exclude;
    }

    // return solution to the current subproblem
    return lookup[key];
}

// Subset Sum Problem
int main()
{
    // Input: a set of items and a sum
    vector<int> A = { 7, 3, 2, 5, 8 };
    int k = 14;

    // total number of items
    int n = A.size();
```

```
54      unordered_map<string, bool> lookup;
55
56      if (subsetSum(A, n - 1, k, lookup)) {
57          cout << "Subsequence with the given sum exists";
58      }
59      else {
60          cout << "Subsequence with the given sum does not exist";
61      }
62
63      return 0;
64  }
```

Download Run Code

Output:

```
Subsequence with the given sum exists
```

Java

Python

The time complexity of the above solution is `O(n × sum)` and requires `O(n × sum)` extra space, where `n` is the size of the input and `sum` is the sum of all elements in the input.

We can also solve this problem in a bottom-up manner. In the bottom-up approach, we solve smaller subproblems first, then solve larger subproblems from them. The following bottom-up approach computes `T[i][j]`, for each `1 <= i <= n` and `1 <= j <= sum`, which is true if subset with sum `j` can be found using items up to first `i` items. It uses the value of smaller values `i` and `j` already computed. It has the same asymptotic runtime as Memoization but no recursion overhead.

Following is the C++, Java, and Python implementation of the idea:

C++

```
1  #include <iostream>
2  #include <vector>
```

```
5  // Returns true if there exists a subsequence of `A` with the given su
6  bool subsetSum(vector<int> const &A, int k)
7  {
8      // total number of items
9      int n = A.size();
10
11     // `T[i][j]` stores true if subset with sum `j` can be attained
12     // using items up to first `i` items
13     bool T[n + 1][k + 1];
14
15     // if 0 items in the list and the sum is non-zero
16     for (int j = 1; j <= k; j++) {
17         T[0][j] = false;
18     }
19
20     // if the sum is zero
21     for (int i = 0; i <= n; i++) {
22         T[i][0] = true;
23     }
24
25     // do for i'th item
26     for (int i = 1; i <= n; i++)
27     {
28         // consider all sum from 1 to sum
29         for (int j = 1; j <= k; j++)
30         {
31             // don't include the i'th element if `j-A[i-1]` is negativ
32             if (A[i - 1] > j) {
33                 T[i][j] = T[i - 1][j];
34             }
35             else {
36                 // find the subset with sum `j` by excluding or includ
37                 T[i][j] = T[i - 1][j] || T[i - 1][j - A[i - 1]];
38             }
39         }
40     }
41
42     // return maximum value
43     return T[n][k];
44 }
45
46 // Subset Sum Problem
47 int main()
48 {
49     // Input: a set of items and a sum
50     vector<int> A = { 7, 3, 2, 5, 8 };
51     int k = 18;
52
53     if (subsetSum(A, k)) {
54         cout << "Subsequence with the given sum exists";
55     }
56     else {
57         cout << "Subsequence with the given sum does not exist";
58     }
59
```

Download Run Code

Output:

```
Subsequence with the given sum exists
```

Java

Python

Array, Dynamic Programming

Algorithm, Amazon, Bottom-up, Medium, Recursive, Top-down