

Grokking the Coding Interview: Patterns for Coding Questions

89% completed

 Search Course

Introduction

Pattern: Sliding Window

Pattern: Two Pointers

Pattern: Fast & Slow pointers

Pattern: Merge Intervals

Pattern: Cyclic Sort

Pattern: In-place Reversal of a LinkedList

Pattern: Tree Breadth First Search

Pattern: Tree Depth First Search

Pattern: Two Heaps

Pattern: Subsets

Pattern: Modified Binary Search

Pattern: Bitwise XOR

Pattern: Top 'K' Elements

Pattern: K-way merge

Pattern : 0/1 Knapsack (Dynamic Programming)

- Introduction
- 0/1 Knapsack (medium)
- Equal Subset Sum Partition (medium)
- Subset Sum (medium)**
- Minimum Subset Sum Difference (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2

Subset Sum (medium)

We'll cover the following

- Problem Statement
- *
- Example 1:
- Example 2:
- Example 3:
- Basic Solution
- Bottom-up Dynamic Programming
- Code
- Time and Space complexity
- Challenge

Problem Statement

Given a set of positive numbers, determine if a subset exists whose sum is equal to a given number 'S'.

Example 1:

```
Input: {1, 2, 3, 7}, S=6
Output: True
The given set has a subset whose sum is '6': {1, 2, 3}
```

Example 2:

```
Input: {1, 2, 7, 1, 5}, S=10
Output: True
The given set has a subset whose sum is '10': {1, 2, 7}
```

Example 3:

```
Input: {1, 3, 4, 8}, S=6
Output: False
The given set does not have any subset whose sum is equal to '6'.
```

Basic Solution

This problem follows the [0/1 Knapsack pattern](#) and is quite similar to [Equal Subset Sum Partition](#). A basic brute-force solution could be to try all subsets of the given numbers to see if any set has a sum equal to 'S'.

So our brute-force algorithm will look like:

```
1 for each number 'i'
2   create a new set which INCLUDES number 'i' if it does not exceed 'S', and recursively
3     process the remaining numbers
4   create a new set WITHOUT number 'i', and recursively process the remaining numbers
5   return true if any of the above two sets has a sum equal to 'S', otherwise return false
```

Since this problem is quite similar to [Equal Subset Sum Partition](#), let's jump directly to the bottom-up dynamic programming solution.

Bottom-up Dynamic Programming

We'll try to find if we can make all possible sums with every subset to populate the array `dp[TotalNumbers][S+1]`.

For every possible sum 's' (where $0 \leq s \leq S$), 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 above two scenarios returns true, we can find a subset with a sum equal to 's'.

Let's draw this visually, with the example input {1, 2, 3, 7}, and start with our base case of size zero:

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

Pattern: Topological Sort (Graph)

- Introduction
- Topological Sort (medium)
- Tasks Scheduling (medium)
- Tasks Scheduling Order (medium)
- All Tasks Scheduling Orders (hard)
- Alien Dictionary (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2

Miscellaneous

- Kth Smallest Number (hard)

Conclusions

- Where to Go from Here

[Mark Course as Completed](#)

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

With only one number, we can form a subset only when the required sum is equal to that number

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

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

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

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

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

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

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

sum: 4-6 index:1=> (dp[index-1][sum] || dp[index-1][sum-2])

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

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

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

sum: 4, index:2=> (dp[index-1][sum] || dp[index-1][sum-3])

8 of 10

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

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

9 of 10

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

sum: 1-6, index:3=> (dp[index-1][sum] || dp[index-1][sum-7])

10 of 10



Code

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

Java
Python3
C++
JS

```

1 const canPartition = function(num, sum) {
2   const n = num.length;
3   const dp = Array(n)
4     .fill(false)
5     .map(() => Array(sum + 1).fill(false));
6
7   // populate the sum=0 columns, as we can always form '0' sum with an empty set
8   for (let i = 0; i < n; i++) dp[i][0] = true;
9
10  // with only one number, we can form a subset only when the required sum is equal to its value
11  for (let s = 1; s <= sum; s++) dp[0][s] = num[0] === s;
12
13  // process all subsets for all sums
14  for (let i = 1; i < num.length; i++) {
15    for (let s = 1; s <= sum; s++) {
16      // if we can get the sum 's' without the number at index 'i'
17      if (dp[i - 1][s]) {
18        dp[i][s] = dp[i - 1][s];
19      } else if (s >= num[i]) {
20        // else include the number and see if we can find a subset to get the remaining sum
21        dp[i][s] = dp[i - 1][s - num[i]];
22      }
23    }
24  }
25
26  // the bottom-right corner will have our answer.
27  return dp[num.length - 1][sum];
28 };

```

RUN
SAVE
RESET
Close

Output

```
true
true
false
```

1.557s

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 required sum.

Challenge

Can we improve our bottom-up DP solution even further? Can you find an algorithm that has $O(S)$ space complexity?

 Show Hint

JavaPython3C++JS JS

```
1 const canPartition = function (num, sum) {
2     const n = num.length;
3     const dp = Array(sum + 1).fill(false);
4
5     // handle sum=0, as we can always have '0' sum with an empty set
6     dp[0] = true;
7
8     // with only one number, we can have a subset only when the required sum is equal to its value
9     for (let s = 1; s <= sum; s++) {
10         dp[s] = num[0] == s;
11     }
12
13     // process all subsets for all sums
14     for (let i = 1; i < n; i++) {
15         for (let s = sum; s >= 0; s--) {
16             // if dp[s]==true, this means we can get the sum 's' without num[i], hence we can move on to
17             // the next number else we can include num[i] and see if we can find a subset to get the
18             // remaining sum
19             if (!dp[s] && s >= num[i]) {
20                 dp[s] = dp[s - num[i]];
21             }
22         }
23     }
24
25     return dp[sum];
26 };
27
28 console.log(`Can partitioning be done: ---> ${canPartition([1, 2, 3, 4], 6)})`);
```

RUNSAVERESETCOPIE

Output 1.675s

```
Can partitioning be done: ---> true
Can partitioning be done: ---> true
Can partitioning be done: ---> false
```

Close

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. [See how](#) 

 Back

Next 

Equal Subset Sum Partition (medium)

Minimum Subset Sum Difference (hard)

Mark as Completed

 Report an Issue  Ask a Question

 Create