

Grokking the Coding Interview: Patterns for Coding Questions

88% 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

Equal Subset Sum Partition (medium)

We'll cover the following

- Problem Statement
- Try it yourself
- Basic Solution
- Code
- Time and Space complexity
- Top-down Dynamic Programming with Memoization
- Code
- Time and Space complexity
- Bottom-up Dynamic Programming
- Code
- Time and Space complexity

Problem Statement

Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both subsets is equal.

Example 1:

```
Input: {1, 2, 3, 4}
Output: True
Explanation: The given set can be partitioned into two subsets with equal sum: {1, 4} & {2, 3}
```

Example 2:

```
Input: {1, 1, 3, 4, 7}
Output: True
Explanation: The given set can be partitioned into two subsets with equal sum: {1, 3, 4} & {1, 7}
```

Example 3:

```
Input: {2, 3, 4, 6}
Output: False
Explanation: The given set cannot be partitioned into two subsets with equal sum.
```

Try it yourself

This problem looks similar to the 0/1 Knapsack problem. Try solving it before moving on to see the solution:

Java
Python3
JS
C++

```
1 const can_partition = function(num) {
2   // TODO: Write your code here
3   return false;
4 };
5
6 console.log(`Can partition: ${can_partition([1, 2, 3, 4])}`)
7 console.log(`Can partition: ${can_partition([1, 1, 3, 4, 7])}`)
8 console.log(`Can partition: ${can_partition([2, 3, 4, 6])}`)
9
```

RUN
SAVE
RESET

Basic Solution

This problem follows the **0/1 Knapsack pattern**. A basic brute-force solution could be to try all combinations of partitioning the given numbers into two sets to see if any pair of sets has an equal sum.

Assume that S represents the total sum of all the given numbers. Then the two equal subsets must have a sum equal to $S/2$. This essentially transforms our problem to: "Find a subset of the given numbers that has a total sum of $S/2$ ".

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/2', and recursively
3   |   | process the remaining numbers
4   |   | create a new set WITHOUT number 'i', and recursively process the remaining items
5   return true if any of the above sets has a sum equal to 'S/2', otherwise return false
```

Code

Here is the code for the brute-force solution:

Solution Review: Problem Challenge 2

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](#)

Java Python C++ JS

```

1 let canPartition = function(num) {
2     let sum = 0;
3     for (let i = 0; i < num.length; i++) sum += num[i];
4
5     // if 'sum' is an odd number, we can't have two subsets with equal sum
6     if (sum % 2 !== 0) return false;
7
8     return canPartitionRecursive(num, sum / 2, 0);
9 };
10
11 function canPartitionRecursive(num, sum, currentIndex) {
12     // base check
13     if (sum === 0) return true;
14
15     if (num.length === 0 || currentIndex >= num.length) return false;
16
17     // recursive call after choosing the number at the currentIndex
18     // if the number at currentIndex exceeds the sum, we shouldn't process this
19     if (num[currentIndex] >= sum) {
20         if (canPartitionRecursive(num, sum - num[currentIndex], currentIndex + 1)) return true;
21     }
22
23     // recursive call after excluding the number at the currentIndex
24     return canPartitionRecursive(num, sum, currentIndex + 1);
25 }
26
27 console.log('Can partitioning be done: ---> ${canPartition([1, 2, 3, 4])}');
28 console.log('Can partitioning be done: ---> ${canPartition([1, 1, 3, 4, 7])}');

```

[RUN](#) [SAVE](#) [RESET](#) [Close](#)

Output 1.702s

```

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

```

Time and Space complexity

The time complexity of the above algorithm is exponential $O(2^n)$, where 'n' represents the total number. The space complexity is $O(n)$, which will be used to store the recursion stack.

Top-down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping sub-problems. As stated in previous lessons, memoization is when we store the results of all the previously solved sub-problems so we can return the results from memory if we encounter a problem that has already been solved.

Since we need to store the results for every subset and for every possible sum, therefore we will be using a two-dimensional array to store the results of the solved sub-problems. The first dimension of the array will represent different subsets and the second dimension will represent different 'sums' that we can calculate from each subset. These two dimensions of the array can also be inferred from the two changing values (sum and currentIndex) in our recursive function `canPartitionRecursive()`.

Code

Here is the code for Top-down DP with memoization:

Java Python C++ JS

```

1 let canPartition = function(num) {
2     let sum = 0;
3     for (let i = 0; i < num.length; i++) sum += num[i];
4
5     // if 'sum' is an odd number, we can't have two subsets with equal sum
6     if (sum % 2 !== 0) return false;
7
8     const dp = [];
9     return canPartitionRecursive(dp, num, sum / 2, 0);
10 };
11
12 function canPartitionRecursive(dp, num, sum, currentIndex) {
13     // base check
14     if (sum === 0) return true;
15
16     if (num.length === 0 || currentIndex >= num.length) return false;
17
18     dp[currentIndex] = dp[currentIndex] || {};
19     // if we have not already processed a similar problem
20     if (typeof dp[currentIndex][sum] === 'undefined') {
21         // recursive call after choosing the number at the currentIndex
22         // if the number at currentIndex exceeds the sum, we shouldn't process this
23         if (num[currentIndex] <= sum) {
24             if (canPartitionRecursive(dp, num, sum - num[currentIndex], currentIndex + 1)) {
25                 dp[currentIndex][sum] = true;
26             }
27         }
28     }

```

[RUN](#) [SAVE](#) [RESET](#) [Close](#)

```

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

```

1.905s

Time and Space complexity

The above algorithm has the time and space complexity of $O(N * S)$, where 'N' represents total numbers and 'S' is the total sum of all the numbers.

Bottom-up Dynamic Programming

Let's try to populate our `dp[][]` array from the above solution by working in a bottom-up fashion. Essentially, we want to find if we can make all possible sums with every subset. This means, `dp[i][s]` will be 'true' if we can make the sum 's' from the first 'i' numbers.

So, for each number at index 'i' ($0 \leq i < \text{num.length}$) and sum 's' ($0 \leq s \leq S/2$), we have two options:

1. Exclude the number. In this case, we will see if we can get 's' from the subset excluding this number: `dp[i-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[i-1][s-num[i]]`

If either of the two above scenarios is true, we can find a subset of numbers with a sum equal to 's'.

Let's start with our base case of zero capacity:

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

'0' sum can always be found through an empty set

1 of 10

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

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

2 of 10

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

sum: 1, index:1=> (dp[index-1][sum] , as the 'sum' is less than the number at index '1'

3 of 10

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

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

4 of 10

	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T		
{1,2,3}	T					
{1,2,3,4}	T					

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

5 of 10

	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T					
{1,2,3,4}	T					

sum: 4,5, index:1=> (dp[index-1][sum] || dp[index-1][sum-2])

6 of 10

	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T		
{1,2,3,4}	T					

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

7 of 10

	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T	T	
{1,2,3,4}	T					

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

8 of 10

	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T	T	T
{1,2,3,4}	T					

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

9 of 10

	0	1	2	3	4	5
1	T	T	F	F	F	F
{1, 2}	T	T	T	T	F	F
{1,2,3}	T	T	T	T	T	T
{1,2,3,4}	T	T	T	T	T	T

sum: 1-5, index:3=> (dp[index-1][sum] || dp[index-1][sum-4])

10 of 10



From the above visualization, we can clearly see that it is possible to partition the given set into two subsets with equal sums. as shown by bottom-right cell: `dp[3][5] => T`

Code

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

```
Java Python3 C++ JS
1 let canPartition = function(num) {
2     const n = num.length;
3     // find the total sum
4     let sum = 0;
5     for (let i = 0; i < n; i++) sum += num[i];
6
7     // if 'sum' is an odd number, we can't have two subsets with same total
8     if (sum % 2 != 0) return false;
9
10    // we are trying to find a subset of given numbers that has a total sum of 'sum/2'.
11    sum /= 2;
12
13    const dp = Array(n)
14        .fill(false)
15        .map(() => Array(sum + 1).fill(false));
16
17    // populate the sum=0 columns, as we can always form '0' sum with an empty set
18    for (let i = 0; i < n; i++) dp[i][0] = true;
19
20    // with only one number, we can form a subset only when the required sum is equal to its value
21    for (let s = 1; s <= sum; s++) {
22        dp[0][s] = num[0] == s;
23    }
24
25    // process all subsets for all sums
26    for (let i = 1; i < n; i++) {
27        for (let s = 1; s <= sum; s++) {
28            // if we can get the sum 's' without the number at index 'i'
RUN SAVE RESET Close 2.078s
```

The code implements a bottom-up dynamic programming solution for the subset sum problem. It first checks if the total sum is odd, as it cannot be partitioned into two equal halves. If even, it initializes a DP array where each row represents a number from the input array and each column represents a target sum from 0 to the total sum. The first column is filled with true because an empty subset sums up to 0. Then, it iterates through each number and each possible sum, updating the DP array based on whether the current sum can be formed by either including or excluding the current number. Finally, it checks if the last row's last column is true, indicating if a subset with the total sum exists.

Time and Space complexity

The above solution has time and space complexity of $O(N * S)$, where 'N' represents total numbers and 'S' is the total sum of all the numbers.

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

Back

0/1 Knapsack (medium)

Next

Subset Sum (medium)

[Mark as Completed](#)

Report an Issue Ask a Question



Explore



Tracks



My Courses



Edpresso



Refer a Friend



Create