# Algorithms and Data Structures

Exercise Session 7

https://n.ethz.ch/~ahmala/and

**Exercise 7.1** *1-3 subset sums* **(1 point)**.

Let $A[1,\dots,n]$ be an array containing $n$ positive integers, and let $b \in \mathbb{N}$. We want to know if there exists a subset $I \subseteq \{1,2,\dots,n\}$, together with multipliers $c_i \in \{1,3\}, i \in I$ such that:

$$b = \sum_{i \in I} c_i \cdot A[i].$$

If this is possible, we say $b$ is a 1-3 subset sum of $A$. For example, if $A = [16, 4, 2, 7, 11, 1]$ and $b = 61$, we could write $b = 3 \cdot 16 + 4 + 3 \cdot 2 + 3 \cdot 1$.

Describe a DP algorithm that, given an array $A[1,\dots,n]$ of positive integers, and a positive integer $b \in \mathbb{N}$ returns True if and only if $b$ is a 1-3 subset sum of $A$. Your algorithm should have asymptotic runtime complexity at most $O(b \cdot n)$.

**Exercise 7.2** *Road trip.*

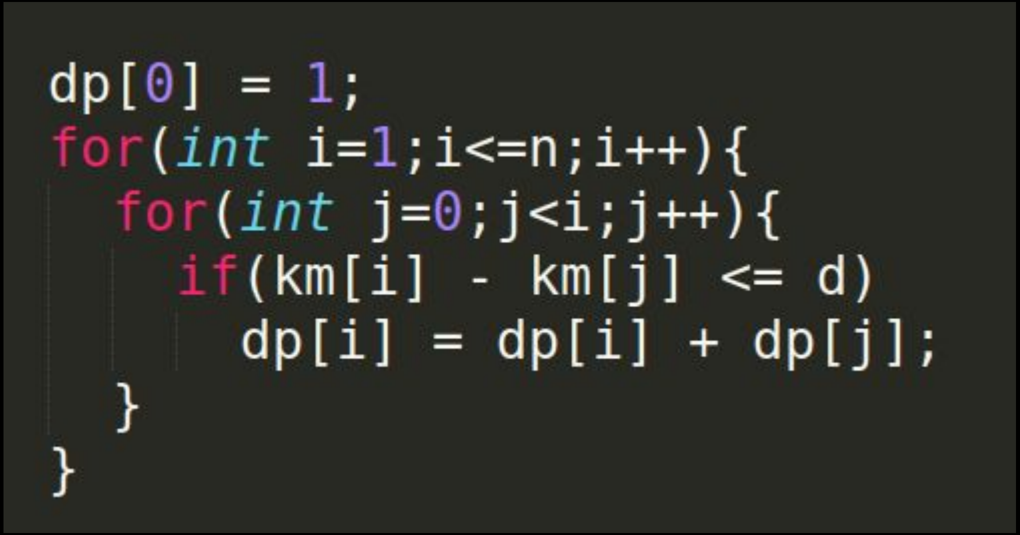
You are planning a road trip for your summer holidays. You want to start from city $C_0$, and follow the only road that goes to city $C_n$ from there. On this road from $C_0$ to $C_n$, there are $n-1$ other cities $C_1, \dots, C_{n-1}$ that you would be interested in visiting (all cities $C_1, \dots, C_{n-1}$ are on the road from $C_0$ to $C_n$). For each $0 \leq i \leq n$, the city $C_i$ is at kilometer $k_i$ of the road for some given $0 = k_0 < k_1 < \dots < k_{n-1} < k_n$.

You want to decide in which cities among $C_1, \dots, C_{n-1}$ you will make an additional stop (you will stop in $C_0$ and $C_n$ anyway). However, you do not want to drive more than $d$ kilometers without making a stop in some city, for some given value $d > 0$ (we assume that $k_i < k_{i-1} + d$ for all $i \in [n]$ so that this is satisfiable), and you also don't want to travel backwards (so from some city $C_i$ you can only go forward to cities $C_j$ with $j > i$).

(a) Provide a *dynamic programming* algorithm that computes the number of possible routes from $C_0$ to $C_n$ that satisfy these conditions, i.e., the number of allowed subsets of stop-cities. Your algorithm should have $O(n^2)$ runtime.

Address the same six aspects as in Exercise 7.1 in your solution.

(b) If you know that $k_i > k_{i-1} + d/10$ for every $i \in [n]$, how can you turn the above algorithm into a linear time algorithm (i.e., an algorithm that has $O(n)$ runtime) ?

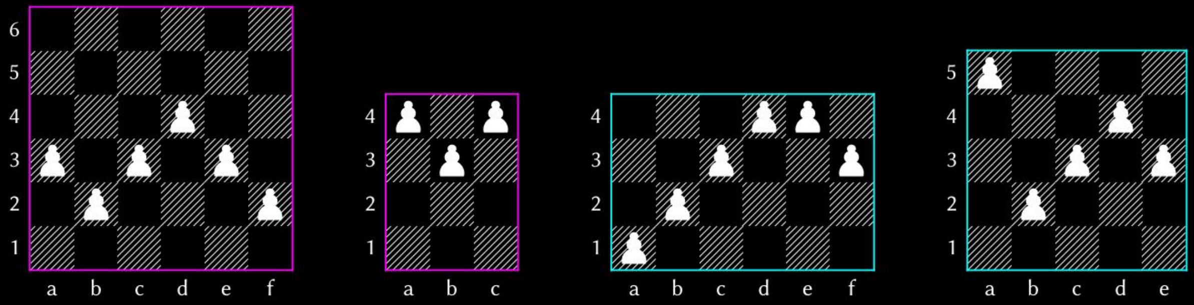


```
dp[0] = 1;
for(int i=1;i<=n;i++){
  for(int j=0;j<i;j++){
    if(km[i] - km[j] <= d)
      dp[i] = dp[i] + dp[j];
  }
}
```

**Exercise 7.3** *Safe pawn lines.*

On an $N \times M$ chessboard ($N$ being the number of rows and $M$ the number of columns), a *safe pawn line* is a set of $M$ pawns with exactly one pawn per column of the chessboard, and such that every two pawns from adjacent columns are located diagonally to each other. When a pawn line is not safe, it is called *unsafe*.

The first two chessboards below show safe pawn lines, the latter two unsafe ones. The line on the third chessboard is unsafe because pawns d4 and e4 are located on the same row (rather than diagonally); the line on the fourth chessboard is unsafe because pawn a5 has no diagonal neighbor at all.



Describe a DP algorithm that, given $N, M > 0$, counts the number of safe pawn lines on an $N \times M$ chessboard. In your solution, address the same six aspects as in Exercise 7.1. Your solution should have complexity at most $O(NM)$.

**Exercise 7.4** *String counting* **(1 point)**.

Given a binary string $S \in \{0,1\}^n$ of length $n$, let $f(S)$ be the number of times "11" occurs in the string, i.e. the number of times a 1 is followed by another 1. In particular, the occurrences do not need to be disjoint. For example $f(\text{“}\underline{1}\underline{1}10\underline{1}1\text{”}) = 3$ because the string contains three 1 that are followed by another 1 (underlined). Given $n$ and $k$, the goal is to count the number of binary strings $S$ of length $n$ with $f(S) = k$.

Describe a DP algorithm that, given positive integers $n$ and $k$ with $k < n$, reports the required number. In your solution, address the same six aspects as in Exercise 7.1. Your solution should have complexity at most $O(nk)$.

***Hint:*** *Define a three dimensional DP table* $DP[1 \ldots n][0 \ldots k][0 \ldots 1]$.

***Hint:*** *The entry* $DP[i][j][l]$ *counts the number of strings of length* $i$ *with* $j$ *occurrences of* "11" *that end in* $l$ *(where* $1 \leq i \leq n, 0 \leq j \leq k$ *and* $0 \leq l \leq 1$*).*

**Exercise 7.5** *Approximately solving knapsack* **(1 point)**.

Consider a knapsack problem with $n$ items with values $v_i \in \mathbb{N}$ and weights $w_i \in \mathbb{N}$ for $i \in \{1, 2, \ldots, n\}$, and weight limit $W \in \mathbb{N}$. Assume[1] that $w_{\max} := \max_{1 \leq i \leq n} w_i \leq W$ and also that $W \leq \sum_{i=1}^{n} w_i$.

For a set of items $I \subseteq \{1, 2, \ldots, n\}$, we write $v(I) = \sum_{i \in I} v_i$ and $w(I) = \sum_{i \in I} w_i$ for the total value (resp. weight) of $I$. So, the solution to a knapsack problem is given by

$$\text{opt} := \max \left\{ v(I) : I \subseteq \{1, 2, \ldots, n\}, \quad w(I) \leq W \right\}.$$

Let $\varepsilon > 0$. We say a set of items $I \subseteq \{1, 2, \ldots, n\}$ is $\varepsilon$-*feasible* if it violates the weight limit by at most a factor $1 + \varepsilon$, that is, if

$$w(I) \leq (1 + \varepsilon) \cdot W.$$

In this exercise, we construct an algorithm that finds an $\varepsilon$-feasible set $I$ with $v(I) \geq \text{opt}$ in polynomial time in $n$ and $1/\varepsilon$.

(a) Let $k \in \mathbb{N}$. Consider a 'rounded version' of the knapsack problem above obtained by replacing the weights $w_i$ by:

$$\widetilde{w_i} := k \cdot \left\lfloor \frac{w_i}{k} \right\rfloor .$$

Recall the dynamical programming algorithm you have seen in the lecture which solves the knapsack problem in time $O(n \cdot W)$. Explain how to modify this algorithm to solve the 'rounded version' in time $O\big((W/k) \cdot n\big)$. For simplicity, you may assume that $W$ is an integer multiple of $k$ for this part.

***Hint:*** *After rounding, all the $\widetilde{w_i}$ are integer multiples of $k$. Use this to reduce the number of table entries you have to compute in the dynamical programming algorithm.*

## Typo in the official solution!!

**Solution:**

The dynamical programming algorithm from the lecture fills out a table $\mathrm{MW}[0,\dots,n][0,\dots,W]$ with entries given by

$$\mathrm{MW}(i,w) := \text{‘largest value } v(I) \text{ one can obtain with } I \subseteq \{1,2,\dots,i\} \text{ and } w(I) \leq w\text{’.}$$

This table is filled using the recursion:

$$\mathrm{MW}(i,w) = \max\left\{\mathrm{MW}(i-1,w),\ v_i + \mathrm{MW}(i, w-w_i)\right\}$$

Note that after rounding, all the $\widetilde{w_i}$ are integer multiples of $k$. Therefore, in the rounded problem, it suffices to only compute the table entries where the index $w$ is an integer multiple of $k$. This leads to $(W/k+1)\cdot(n+1)$ table entries in total, each of which takes constant time to compute, meaning runtime $O\big((W/k)\cdot n\big)$ in total.

We write $\mathrm{opt}_k$ for the optimum solution value of the rounded problem in part (a). From now on, you may assume that your modified algorithm also returns a set $I_k$ with $v(I_k) = \mathrm{opt}_k$ and $\sum_{i \in I_k} \widetilde{w_i} \leq W$.

(b) Explain why $\mathrm{opt}_k \geq \mathrm{opt}$.

(c) Set $\varepsilon := (nk)/w_{\max}$. Show that $w(I_k) \leq (1+\varepsilon) \cdot W$, that is, show that $I_k$ is $\varepsilon$-feasible.

***Hint:*** *Show that* $w_i \leq \widetilde{w_i} + k$ *for each* $i \in \{1, 2, \ldots, n\}$. *We know that* $\sum_{i \in I_k} \widetilde{w_i} \leq W$. *Combine these facts to show that* $w(I_k) := \sum_{i \in I_k} w_i \leq W + n \cdot k$. *Finally, use the fact that* $W/w_{\max} \geq 1$ *to conclude your proof.*

(d) Now let $\varepsilon > 0$ be arbitrary. Describe an algorithm that finds an $\varepsilon$-feasible set $I$ of items with $v(I) \geq \text{opt}$ in time $O(n^3/\varepsilon)$. Prove the runtime guarantee and correctness of your algorithm.

***Hint:*** *Apply the algorithm of part (a) to the rounded problem with $k = (w_{\max} \cdot \varepsilon)/n$. For simplicity, you may assume that this $k$ is an integer in your proof. Then, use the assumption that $W \leq \sum_{i=1}^{n} w_i$ $(\leq n \cdot w_{\max})$ to bound the runtime of the algorithm in terms of $n$ and $1/\varepsilon$. Finally, use part (b) and part (c) to show correctness.*

(e)* Let $\varepsilon = 1/100$. Give an example of a knapsack problem which has an $\varepsilon$-feasible solution $I$ with value

$$v(I) = 2 \cdot \mathrm{opt},$$

Your example should satisfy $w_{\max} \leq W$ and $W \leq \sum_{i=1}^{n} w_i$.

(e)* Let $\varepsilon = 1/100$. Give an example of a knapsack problem which has an $\varepsilon$-feasible solution $I$ with value

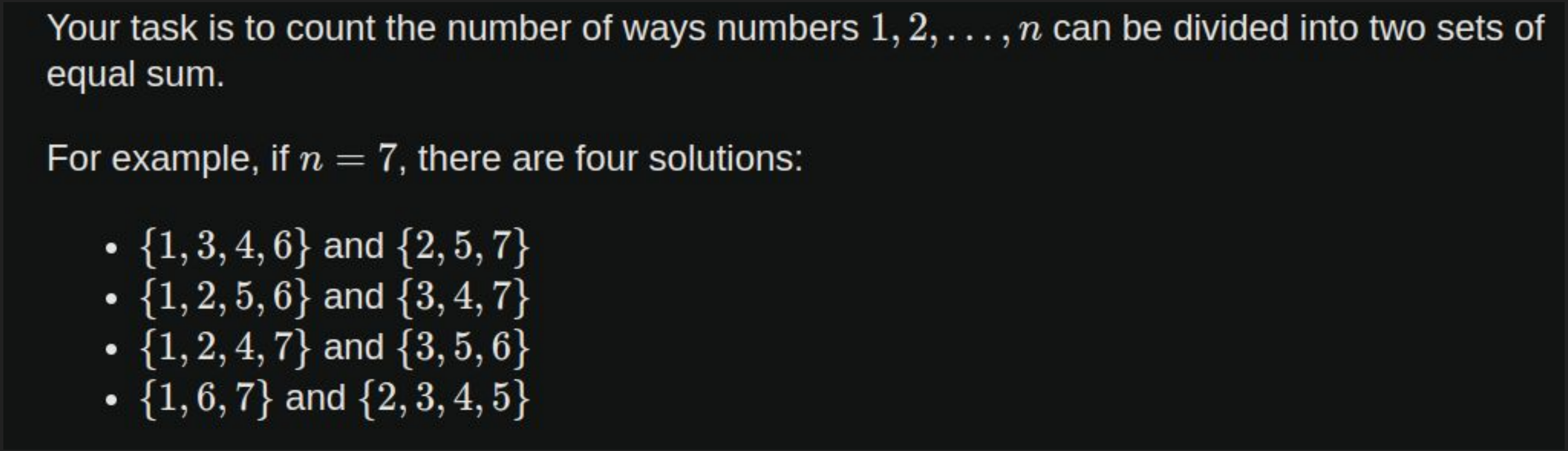
$$v(I) = 2 \cdot \mathrm{opt},$$

Your example should satisfy $w_{\max} \leq W$ and $W \leq \sum_{i=1}^{n} w_i$.

**Solution:**

Consider the knapsack problem with two items, $v_1 = v_2 = w_1 = w_2 = 101$ and $W = 200$. Then, $\mathrm{opt} = 101$ as we can only take one item. However, taking both items yields a solution which violates the weight limit by only a factor $1/100$ (as $(202 - 200)/200 = 1/100$). Thus there is an $\varepsilon$-feasible solution with value $202 = 2 \cdot \mathrm{opt}$.

## DP Exercise



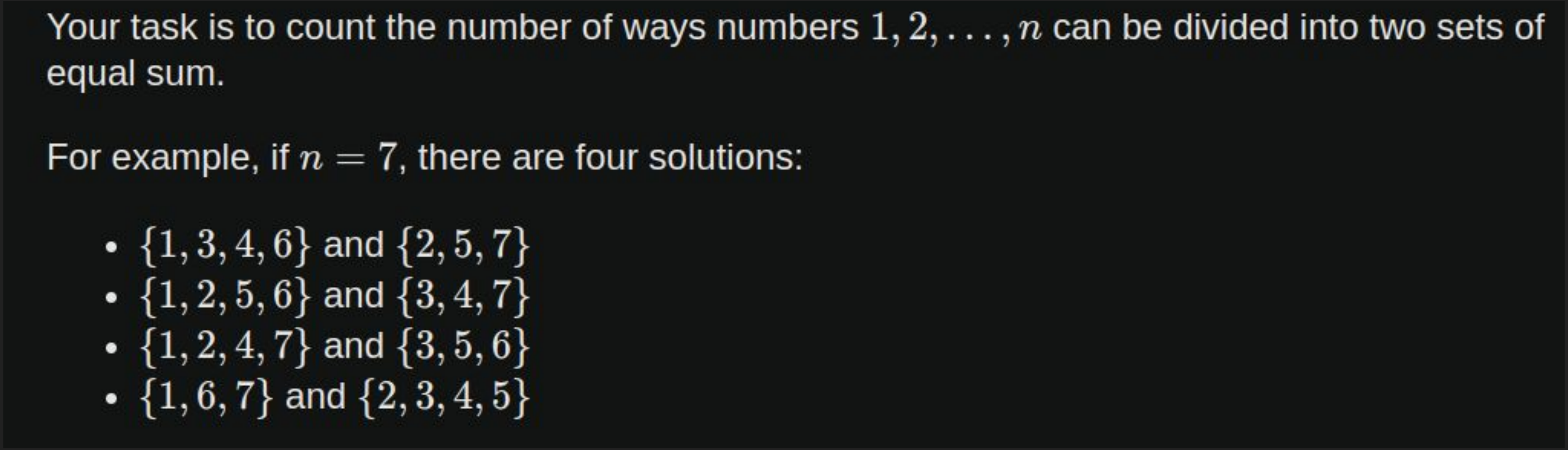
Your task is to count the number of ways numbers $1, 2, \ldots, n$ can be divided into two sets of equal sum.

For example, if $n = 7$, there are four solutions:

- $\{1, 3, 4, 6\}$ and $\{2, 5, 7\}$
- $\{1, 2, 5, 6\}$ and $\{3, 4, 7\}$
- $\{1, 2, 4, 7\}$ and $\{3, 5, 6\}$
- $\{1, 6, 7\}$ and $\{2, 3, 4, 5\}$

O(n^3) solution expected!

Source

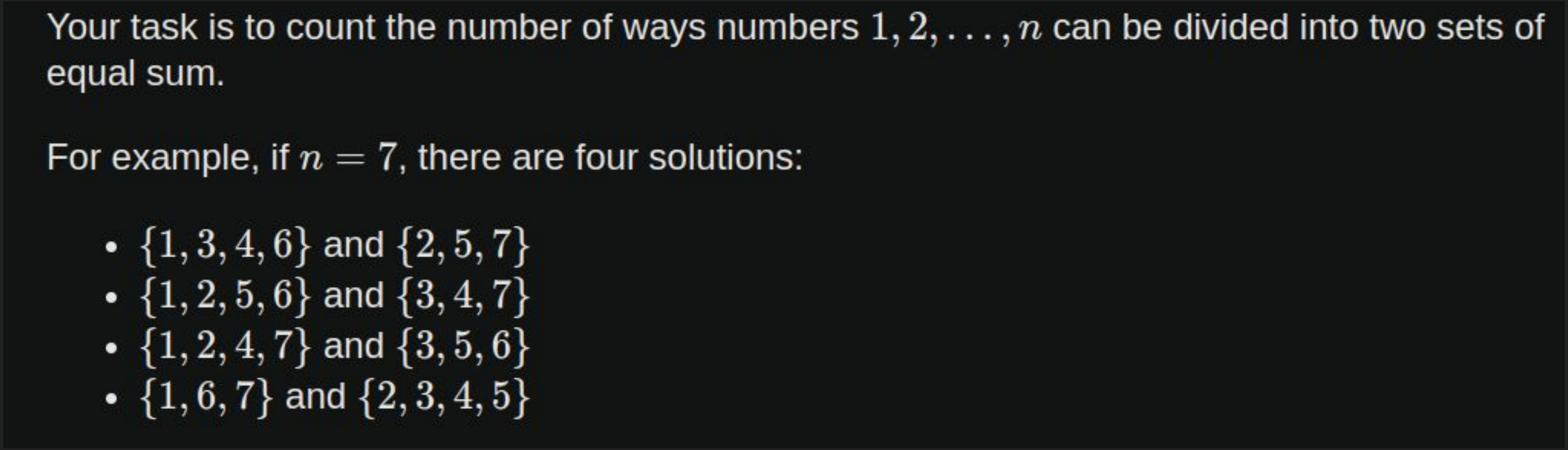

Your task is to count the number of ways numbers $1, 2, \ldots, n$ can be divided into two sets of equal sum.

For example, if $n = 7$, there are four solutions:

- $\{1, 3, 4, 6\}$ and $\{2, 5, 7\}$
- $\{1, 2, 5, 6\}$ and $\{3, 4, 7\}$
- $\{1, 2, 4, 7\}$ and $\{3, 5, 6\}$
- $\{1, 6, 7\}$ and $\{2, 3, 4, 5\}$

dp[i][x] = number of ways to make sum x using subsets of the numbers 1..i

Source



Your task is to count the number of ways numbers $1, 2, \ldots, n$ can be divided into two sets of equal sum.

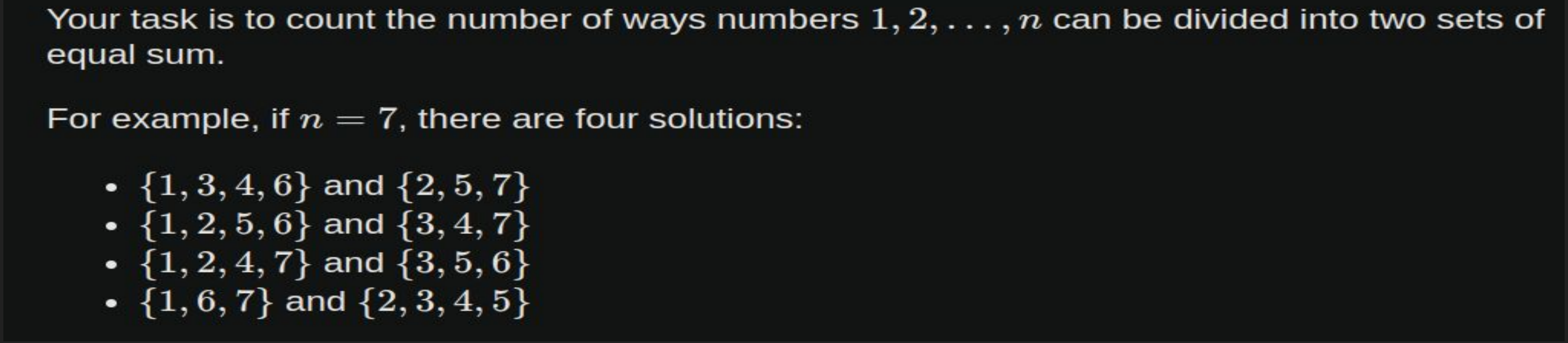
For example, if $n = 7$, there are four solutions:

- $\{1, 3, 4, 6\}$ and $\{2, 5, 7\}$
- $\{1, 2, 5, 6\}$ and $\{3, 4, 7\}$
- $\{1, 2, 4, 7\}$ and $\{3, 5, 6\}$
- $\{1, 6, 7\}$ and $\{2, 3, 4, 5\}$

dp[i][x] = number of ways to make sum x using subsets of the numbers 1..i

dp[i][x] = dp[i-1][x] + dp[i-1][x-i]

Source


Your task is to count the number of ways numbers $1, 2, \ldots, n$ can be divided into two sets of equal sum.

For example, if $n = 7$, there are four solutions:

- $\{1, 3, 4, 6\}$ and $\{2, 5, 7\}$
- $\{1, 2, 5, 6\}$ and $\{3, 4, 7\}$
- $\{1, 2, 4, 7\}$ and $\{3, 5, 6\}$
- $\{1, 6, 7\}$ and $\{2, 3, 4, 5\}$

dp[i][x] = number of ways to make sum x using subsets of the numbers 1..i

dp[i][x] = dp[i-1][x] + dp[i-1][x-i]
The answer is stored in dp[7][14]/2 or dp[6][14] for n=7

Since you aren't having 'n' in your first set, you'll never have a first set that consists of a combination involving 'n' i.e. all the times a combination requires 'n' you are simply not counting it... thus you end up counting things only once.
Source

## Peer Grading(with the former groups)

Exercise 7.1

While sending to me please include the group you received their work in cc.

https://docs.google.com/spreadsheets/d/1owPsJsd9THBWInwFcVjKCc0f_r6n4pGwwDKMDdwaCjM/edit?usp=sharing