# day02

December 2, 2023

## 1 Day 02

### 1.1 Cube Conundrum

You're launched high into the atmosphere! The apex of your trajectory just barely reaches the surface of a large island floating in the sky. You gently land in a fluffy pile of leaves. It's quite cold, but you don't see much snow. An Elf runs over to greet you.

The Elf explains that you've arrived at **Snow Island** and apologizes for the lack of snow. He'll be happy to explain the situation, but it's a bit of a walk, so you have some time. They don't get many visitors up here; would you like to play a game in the meantime?

As you walk, the Elf shows you a small bag and some cubes which are either red, green, or blue. Each time you play this game, he will hide a secret number of cubes of each color in the bag, and your goal is to figure out information about the number of cubes.

To get information, once a bag has been loaded with cubes, the Elf will reach into the bag, grab a handful of random cubes, show them to you, and then put them back in the bag. He'll do this a few times per game.

You play several games and record the information from each game (your puzzle input). Each game is listed with its ID number (like the 11 in Game 11: …) followed by a semicolon-separated list of subsets of cubes that were revealed from the bag (like 3 red, 5 green, 4 blue).

For example, the record of a few games might look like this:

```
Game 1: 3 blue, 4 red; 1 red, 2 green, 6 blue; 2 green
Game 2: 1 blue, 2 green; 3 green, 4 blue, 1 red; 1 green, 1 blue
Game 3: 8 green, 6 blue, 20 red; 5 blue, 4 red, 13 green; 5 green, 1 red
Game 4: 1 green, 3 red, 6 blue; 3 green, 6 red; 3 green, 15 blue, 14 red
Game 5: 6 red, 1 blue, 3 green; 2 blue, 1 red, 2 green
```

In game 1, three sets of cubes are revealed from the bag (and then put back again). The first set is 3 blue cubes and 4 red cubes; the second set is 1 red cube, 2 green cubes, and 6 blue cubes; the third set is only 2 green cubes.

The Elf would first like to know which games would have been possible if the bag contained **only 12 red cubes, 13 green cubes, and 14 blue cubes**?

In the example above, games 1, 2, and 5 would have been **possible** if the bag had been loaded with that configuration. However, game 3 would have been **impossible** because at one point the Elf showed you 20 red cubes at once; similarly, game 4 would also have been **impossible** because

the Elf showed you 15 blue cubes at once. If you add up the IDs of the games that would have been possible, you get **8**.

Determine which games would have been possible if the bag had been loaded with only 12 red cubes, 13 green cubes, and 14 blue cubes. **What is the sum of the IDs of those games**?

[1]:
```
## Python Imports

from collections import defaultdict
from math import prod
from pathlib import Path
```

## Part One

It seems like this is another straightforward solution: games are impossible if the number of dice drawn for a colour is greater than the number of cubes of that colour which are available. Most of the code handles parsing the flattened game structure into a form that we can calculate on.

I load the data into a (`defaultdict`) dictionary where the game number is the key, and the value is a list of individual draw outcomes from the gate. Each draw outcome is a `dict` keyed by colour, with value the number of cubes drawn.

To find valid games, I iterate over each game and initially assume it is valid. I test that assumption by iterating over each draw outcome and checking the number of drawn cubes of each colour against the number of available cubes. If the number of drawn cubes is greater than those available, the game is impossible, and I mark it as invalid and move on to the next one. If the game is not found to be invalid, it's added to the valid list, and the list of valid games is returned.

[2]:
```
def read_data(path):
    games = defaultdict(list)  # dict of list of dicts representing game␣
 ↪outcomes
    with path.open() as ifh:
        for game, line in enumerate([_.strip() for _ in ifh.readlines()]):
            outcomes = [_.split(", ") for _ in line.split(": ")[-1].split("; ")]
            for outcome in outcomes:
                outcomedict = {}  # dict keyed by colour, with number of dice␣
 ↪drawn
                for cubes in outcome:
                    val, colour = cubes.split()
                    outcomedict[colour] = int(val)
                games[game + 1].append(outcomedict)
    return games

def find_games(cubes, games):
    valid_games = []  # IDs of valid games, given the cubes constraint
    for game, outcomes in games.items():
        validgame = True  # initial assumption: the game is valid
        for outcome in outcomes:
            for colour, val in outcome.items():
```

```
                if cubes[colour] < val:  # cube constraint could not satisfy␣
↪draw
                    validgame = False  # the game is invalid
                    break
        if validgame == True:  # the game is valid within cube constraints
            valid_games.append(game)
    return valid_games
```

[3]:
```
cubes = {"red": 12, "green": 13, "blue": 14}  # The number of available cubes␣
↪for each colour

games = read_data(Path("data/day02_test.txt"))
sum(find_games(cubes, games))
```

[3]: 8

[4]:
```
games = read_data(Path("data/day02_data.txt"))
sum(find_games(cubes, games))
```

[4]: 2551

### 1.2 Part Two

The Elf says they've stopped producing snow because they aren't getting any **water**! He isn't sure why the water stopped; however, he can show you how to get to the water source to check it out for yourself. It's just up ahead!

As you continue your walk, the Elf poses a second question: in each game you played, what is the **fewest number of cubes of each color** that could have been in the bag to make the game possible?

Again consider the example games from earlier:

```
Game 1: 3 blue, 4 red; 1 red, 2 green, 6 blue; 2 green
Game 2: 1 blue, 2 green; 3 green, 4 blue, 1 red; 1 green, 1 blue
Game 3: 8 green, 6 blue, 20 red; 5 blue, 4 red, 13 green; 5 green, 1 red
Game 4: 1 green, 3 red, 6 blue; 3 green, 6 red; 3 green, 15 blue, 14 red
Game 5: 6 red, 1 blue, 3 green; 2 blue, 1 red, 2 green
```

```
In game 1, the game could have been played with as few as 4 red, 2 green, and 6 blue cubes. If
Game 2 could have been played with a minimum of 1 red, 3 green, and 4 blue cubes.
Game 3 must have been played with at least 20 red, 13 green, and 6 blue cubes.
Game 4 required at least 14 red, 3 green, and 15 blue cubes.
Game 5 needed no fewer than 6 red, 3 green, and 2 blue cubes in the bag.
```

The **power** of a set of cubes is equal to the numbers of red, green, and blue cubes multiplied together. The power of the minimum set of cubes in game 1 is 48. In games 2-5 it was 12, 1560, 630, and 36, respectively. Adding up these five powers produces the sum **2286**.

For each game, find the minimum set of cubes that must have been present. **What is the sum of the power of these sets**?

For the second part, we need to find the maximum number of cubes drawn for each colour, in a game. This would have also been a useful part of the solution to Part One (if the number of cubes specified in the constraint for a colour is less than the minimum required for a game, the game is invalid). Once we have this list, calculating the power is straightforward.

Below, my solution iterates over each game, storing the maximum number of cubes for each colour in a `dict`. The first outcome of a game is the initial maximum, and the remaining outcomes are considered in turn, updating the maximum for a colour if that draw requires more cubes. The function returns a `dict` keyed by game ID, with value the minimum requirement by colour as a `dict`.

I used a function to calculate the powers for this `dict` of minimum requirements and return a `list` of the individual powers.

[5]:
```
def min_cubes(games):
    game_mincubes = {}
    for game, outcomes in games.items():
        mincubes = outcomes[0]
        for outcome in outcomes[1:]:
            for colour, val in outcome.items():
                if colour not in mincubes:
                    mincubes[colour] = val
                else:
                    mincubes[colour] = max(mincubes[colour], val)
        game_mincubes[game] = mincubes
    return game_mincubes

def calc_powers(mincubes):
    powers = []
    for game, cubes in mincubes.items():
        powers.append(prod(cubes.values()))
    return powers
```

[6]:
```
games = read_data(Path("data/day02_test.txt"))
mincubes = min_cubes(games)
sum(calc_powers(mincubes))
```

[6]: 2286

[7]:
```
games = read_data(Path("data/day02_data.txt"))
mincubes = min_cubes(games)
sum(calc_powers(mincubes))
```

[7]: 62811

[ ]: