Solution Combinatorics

January 30, 2025

1 Solution Combinatorics

Arthur Ryman, last updated 2025-01-29

1.1 Introduction

This notebook shows how to compute the total number of combinations of the Instant Insanity puzzle. The branch of mathematics that focuses on how to count combinations is called *Combinatorics*. The number of combinations for any puzzle is very interesting since it indicates how hard it is to solve the puzzle.

1.2 41,472 or 82,944?

1.2.1 41,472

The 1947 Eureka paper The Coloured Cubes Problem by F. de Carteblanche which gives the elegant graph theory solution to the problem states that if you pick a random arrangement of the cubes your chance of it being a solution is 1 in 41,472. At that time, the puzzle was called the *Tantalizer*. Carteblanche was the pseudonym of a group of four Cambridge mathematics students which included the eminent graph theorist Bill Tutte.

1.2.2 82,944

The 2008 Gresham College lecture The Four Cubes Problem by Robin Wilson, Gresham Professor of Geometry, gives the number as 82,944.

The 2018 PBS Infinite Series video Instant Insanity Puzzle by Tai-Danea Bradley also gives the number as 82,944.

The 2024 Winning Solutions Instant Insanity package similarly advertises 82,944 combinations.

1.2.3 Who's Correct?

Let's apply some combinatorics and compute the answer for ourselves.

1.3 Counting the Combinations

First we need to precisely define what we mean by a combination of the cubes. For example, suppose we put the four cubes in a bag, draw them out one at a time, and place them in a horizontal row without looking. So we are picking a random order for the cubes and a random orientation for each cube.

(animate this)

1.3.1 Counting the Number of Cube Orderings

One of the most basic theorems of combinatorics is that if you have a set of n distinct objects then you can order them in

$$n \times (n-1) \times \cdots \times 2 \times 1$$

ways. This quantity is denoted $n!$ which is spoken as n *factorial*. To see this observe that you have n ways to choose the first object. Now there are $n-1$ objects remaining so for each of the n first objects there are $n-1$ ways to choose the second object. Therefore there are $n \times (n-1)$ orderings of two object. Continuing this reasoning, we get that there are $n!$ orderings n objects.

Let's compute 4!.

[1]:
```
number_of_cube_orderings = 4 * 3 * 2 * 1

print(number_of_cube_orderings)
```

```
24
```

However, do we really care about the order of the cubes? If we have found one solution then we can produce another solution simply by swapping the positions of the cubes without changing their orientations. That seems like artificially inflating the number of solutions. Two solutions that differ simply in the order of the cubes should not be considered as distinct solutions. Therefore, let's simply pick an order for the cubes.

As we'll discuss later, swapping the positions of the cubes is an example of a symmetry operation on the set of solutions since it sends solutions to solutions. In general, we won't consider solutions that are related by symmetries to be essentially distinct.

1.3.2 Counting the Number of Cube Orientations

Given a fixed order for the cubes, our remaining freedom is in orienting each cube. We can specify an orientation by saying which face is on top and which face is in front. Let's count the number of orientations.

We'll use another basic theorem of combinatorics. If we have two sets of things, say X containing n things and Y containing m things then the number of ways we can pair things from X with things from Y is $n \times m$.

A cube has 6 faces. Therefore, there are 6 ways to pick the top face.

(animate this)

[2]:
```
number_of_top_face_choices = 6
```

Having picked the top face, we can spin the cube one quarter turn about its vertical axis to pick the front face.

(animate this)

[3]:
```
number_of_front_face_choices = 4
```

Therefore, the total number of cube orientations is 6×4.

[4]:
```
number_of_cube_orientations = number_of_top_face_choices *␣
↪number_of_front_face_choices

print(number_of_cube_orientations)
```

```
24
```

1.3.3 Counting the Number of Cube Arrangements

We have four cubes and we can pick any one of the 24 orientations for each cube. Apply the basic combination rule again to get the total number of cube arrangements. Multiply the number of cube orientations by itself four times. This is the same as raising the number of cube orientations to the power 4.

[5]:
```
number_of_cube_arrangements = number_of_cube_orientations ** 4

print(number_of_cube_arrangements)
```

```
331776
```

We get 331,776 which is much bigger than 82,944. Why the difference? The reason is that we should regard some arrangements as being essentially the same.

Suppose we have a solution. Then we can easily get another solution simply by rotating the row of cubes one quarter of a turn about the horizontal axis. Furthermore, we can do this four times before ending up with the arrangement we started from. Therefore each solution belongs of a family of four essentially equivalent solutions.

(animate this)

Let the symbol Q denote the operation of rotating the row of cubes one quarter of a turn along the horizontal axis. We call Q a *symmetry* operation for the solutions since it sends solutions to solution. We should therefore not regard arrangements that are related by a Q operation as being essentially distinct.

Mathematicians say that arrangements, or mathematical objects in general, that are related by some symmetry are the same *modulo* that symmetry. We therefore need to compute the number of arrangements modulo Q. This can be done by dividing the total number of arrangements by the number of arrangements in each family of equivalent arrangements, which for Q is 4.

[6]:
```
Q_size = 4
number_of_arrangements_modulo_Q = number_of_cube_arrangements // Q_size

print(number_of_arrangements_modulo_Q)
```

```
82944
```

Eureka! Now we understand where the number 82,944 comes from. But what about 41,472? The explanation is similar to the above. We have yet another symmetry.

Suppose we have a solution. We can obtain another solution by rotating each cube by one half turn about its vertical axis. Let's refer to this operations by the symbol H. We can apply the H operation twice before returning to the starting arrangement. Therefore, we need to further divide the number of arrangements by 2 to get the number of essentially distinct arrangements.

[7]:
```
H_size = 2
number_of_arrangements_modulo_QH = number_of_arrangements_modulo_Q // H_size

print(number_of_arrangements_modulo_QH)
```

```
41472
```

Success! We now understand where the number 41,472 comes from. Carteblanche was right!

1.4 Brute-Force Search

We now know how many arrangements, namely 41,472, we'd have to check in order to find a solution. Given enough time and patience, we could systematically generate every possible essentially distinct arrangement and check if it was a solution. Mathematicians call this approach a *brute-force search*.

How long would a brute-force search take? Suppose a human was doing the search. As a rough estimate, suppose it takes 1 second to rotate a cube and 1 second to check if the arrangement is a solution. Let's compute the total time.

[8]:
```
seconds_per_arrangement = 5
total_seconds = number_of_arrangements_modulo_QH * seconds_per_arrangement

print(total_seconds)
```

```
207360
```

[9]:
```
seconds_per_minute = 60
total_minutes = total_seconds / seconds_per_minute

print(total_minutes)
```

```
3456.0
```

[10]:
```
minutes_per_hour = 60
total_hours = total_minutes / minutes_per_hour

print(total_hours)
```

```
57.6
```

[11]:
```
hours_per_day = 24
total_days = total_hours / hours_per_day

print(total_days)
```

```
2.4
```

Therefore, a human would find the solution if they worked nonstop for 2.4 days and made no errors. This is called the *worst case* time since it assumes that you are unlucky and have to generate all the arrangements before you find the solution.

Clearly, this puzzle is challenging which accounts for its popularity. The package states that over 20 million copies have been sold!

1.5 Next

How long would a computer take to find the solution? One of us actually tried this in 1967 using a Fortran 4 program running on their high school's IBM 1130 computer. Next, we'll write some Python code to perform the brute-force search and then optimize it for speed.