**Perfect Numbers**

In Practice Mode

Introduction

Determine if a number is perfect, abundant, or deficient based on Nicomachus' (60 - 120 CE) classification scheme for natural numbers.

The Greek mathematician [Nicomachus](https://en.wikipedia.org/wiki/Nicomachus" \t "_blank) devised a classification scheme for natural numbers, identifying each as belonging uniquely to the categories of **perfect**, **abundant**, or **deficient** based on their [aliquot sum](https://en.wikipedia.org/wiki/Aliquot_sum). The aliquot sum is defined as the sum of the factors of a number not including the number itself. For example, the aliquot sum of 15 is (1 + 3 + 5) = 9

* **Perfect**: aliquot sum = number
  + 6 is a perfect number because (1 + 2 + 3) = 6
  + 28 is a perfect number because (1 + 2 + 4 + 7 + 14) = 28
* **Abundant**: aliquot sum > number
  + 12 is an abundant number because (1 + 2 + 3 + 4 + 6) = 16
  + 24 is an abundant number because (1 + 2 + 3 + 4 + 6 + 8 + 12) = 36
* **Deficient**: aliquot sum < number
  + 8 is a deficient number because (1 + 2 + 4) = 7
  + Prime numbers are deficient

Implement a way to determine whether a given number is **perfect**. Depending on your language track, you may also need to implement a way to determine whether a given number is **abundant** or **deficient**.

Getting Started

Make sure you have read the "Guides" section of the [C track](https://exercism.io/my/tracks/c) on the Exercism site. This covers the basic information on setting up the development environment expected by the exercises.

Passing the Tests

Get the first test compiling, linking and passing by following the [three rules of test-driven development](http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd).

The included makefile can be used to create and run the tests using the test task.

make test

Create just the functions you need to satisfy any compiler errors and get the test to fail. Then write just enough code to get the test to pass. Once you've done that, move onto the next test.

As you progress through the tests, take the time to refactor your implementation for readability and expressiveness and then go on to the next test.

Try to use standard C99 facilities in preference to writing your own low-level algorithms or facilities by hand.