# Goldbach Pair Calculator

## Usage

The Golabach pair calculator has been written as a console application, expecting to take up to two arguments when called.

The first is the number pairs of primes should sum to, and the second is a flag dictating whether to calculate just one pair, or loop through and calculate all pairs.

So for instance:

goldbach.exe 10

Will return

3 & 7

Whereas:

goldbach.exe 10 /all

Will return

3&7  
5&5

Acceptable arguments to produce all pairs are “all” or the abbreviation “a”. The argument option is prefixed by either slash “/” or hyphen “-”.

If arguments are not supplied, the program will prompt for a number, and assume that only one pair is required.

## Solution structure

The solution is formed of three projects: the application itself, a library of classes for generation of primes, validation etc., and a unit test project.

### Goldbach

This project is a simple console application. All it does is collect the input arguments, or request them if not supplied when invoked. It then calls to a validator class to check the input number matches the specified rules. If not met, it reports the failure to the console.

Assuming successful validation, the number is the passed to the pair calculator – along with the “all pairs” option, if supplied. The pairs produced are then reported to the console.

### Goldbach.Data

Goldbach.Data is a library project containing classes and interfaces implementing the core activities solving the problem requires.

A key part is being able to retrieve prime numbers, so there are a set of classes for generating and storing them in memory. The storage has been contrived so that the same prime should not be generated more than once in any given program run.

Another part of the problem is asserting that the number input by a user is applicable to the problem – so a flexible validation mechanism was implemented. Flexibility is offered by the generic implementation of the target type – in the particular instance, it will only be used with integers, but it could be reused to implement validation on floating point numbers, for instance – and by the list of expressions representing the rules to be applied to assert if a value is valid.

A base class implements a “vanilla” application of these rules, and a conversion from a string to the target type – mindful that the target application for this library is a console application.

Some extension methods to assist working with the argument string array passed to the main application are also found in this library project.

Having the library as a separate project allows for highlighting the use of internal constructors, to allow parts of an object to be configured for testing, whilst protecting those parts from actual consumers. This can be achieved by allowing anything with internal access to be seen by a specific assembly – in this case the unit test project.

For instance, the PrimeNumberCache exposes an internal constructor allowing the private list of primes to be initialised. Exposing this to the test project then makes it possible to test for which methods of the prime number generator are called, and when they are called.

### Goldbach.Test

Hopefully self-explanatory, the Goldbach.Test project is a test suite, ensuring that units of work implemented in the various classes are performing their jobs correctly.

The tests are written using the NUnit framework, and Fake It Easy provides mocking – for example removing the need for an actual prime number cache and generator when testing the object calculating Goldbach pairs.

There are a couple of “for fun” tests, which – having tested that it generates the expected prime numbers for lower limits – time how long it takes to generate all the primes up to 100,000,000. It’s a fair while.

## Prime generators and optimisations

Though not strictly part of the requirement, curiosity dictated that different approaches to getting prime numbers should be tried, just to see how they stack up against each other. This is far from a real performance study, and just a bit of intrigue.

### Implemented generators

#### Loop all factors

The initial implementation – BasicPrimeNumberGenerator – simply loops through each number in the requested range, and tries to divide it by every integer less than the square root of the current number. If any results in a remainder free division, then it is not prime, and the next number is tried.

#### Sieve of Eratosthenes

Another implementation – PrimeNumberSieve – is an implementation of the algorithm described here: <https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes>. Essentially, it takes all numbers up to the limit, and then removes any that are multiples of any number below the square root of the limit.

#### Read from disk

The final approach was to just obtain a file with a list of all the primes in it (found here: <http://www.naturalnumbers.org/primes.html>) and read that into memory – PrimeNumberReader.

### Performance

Time didn’t allow a fully detailed analysis – but a set of tests were written against each implementation, timing how long they took to obtain all the primes up to a given limit. There was an assertion that the expected number of primes below that number were found by each. The number of expected primes was found here: <https://primes.utm.edu/howmany.html> (see Table 1). The time taken for each test was measured using the diagnostic stopwatch in the .NET framework.

As stated, time didn’t allow for many runs of the tests, but each implementation was asked to generate the primes up to 100, 1,000, 10,000, 100,000, 1,000,000, 10,000,000, 15,485,863\*, 100,000,000.

\*this is the highest prime in the file obtained for the prime number reader, and therefore where tests for that had to cease. The 100,000,000 case was not run for the sieve prime finding algorithm, since even on the more powerful computer, it took over half an hour, and still hadn’t finished…

The tests were run on two computers, to see how resources affected the performance. The first, a 4.5GHz Intel i7, with 2400MHz DDR3 RAM, and the second, only a 2.5GHz i7, with 1333MHz RAM.

#### Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | Generator | | | Desktop, 4.5GHz i7 |
| Basic | Reader | Sieve |
| Primes up to | 100 | 0 | 0 | 0 |
| 1,000 | 0 | 0 | 0 |
| 10,000 | 1 | 0 | 2 |
| 100,000 | 25 | 2 | 67 |
| 1,000,000 | 505 | 28 | 2761 |
| 10,000,000 | 12582 | 212 | 89730 |
| 15,485,863 | 23368 | 297 | 174320 |
| 100,000,000 | 320938 | N/A | Cancelled |

Table 1 – 4.5GHz i7, 16GB DDR3 @2400MHz

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | Generator | | | Laptop, 2.5GHz i7 |
| Basic | Reader | Sieve |
| Primes up to | 100 | 0 | 0 | 0 |
| 1,000 | 0 | 0 | 0 |
| 10,000 | 2 | 0 | 4 |
| 100,000 | 46 | 14 | 106 |
| 1,000,000 | 893 | 53 | 4356 |
| 10,000,000 | 20154 | 435 | 135507 |
| 15,485,863 | 37466 | 668 | 263633 |
| 100,000,000 | 539225 | N/A | Cancelled |

Table 2 – 2.5GHz i7, 4GB DDR3 @1333MHz

It can be seen from the tables above, that up to a limit of about 10,000, all three methods of obtaining primes offer much the same performance – with the exception that the reader requires a 20MB file to also be distributed. Though that could be reduced to just the prime numbers, and lose the additional metadata.

After this point, the sieving method becomes by far the least efficient, and – aside from the aforementioned storage space – just reading from a file is by far the fastest.