# Assignment overview

Math and computer science geeks sure do love prime numbers. In CS, primes are important in information security, being used in such areas as public-key cryptography.

In this assignment you will be counting (to the best of your computer’s ability) how many prime numbers exist below a given number N. For example, there are 25 prime numbers below 100:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

# Brute force

Here is an *inefficient* way to determine if a particular whole number P is prime. It tests every potential divisor d from 2 to P-1 to see if d goes evenly into P.

**Algorithm** **IsPrime(P)**

if P=0 or P=1

return false

d = 2

while d < P

if P%d = 0

return false

else

d = d+1

return true

end

Here are two easy improvements on that algorithm:

1. (Obvious, sledgehammer-level badness) The loop upper bound of “d<P” is wasteful. There’s no need for d to go higher than sqrt(P). Nothing larger than sqrt(P) could ever go into P.
2. (Slightly more subtle) There is no need to test any even numbers except d=2 itself. If P is divisible by some even number, then it’s also going to be divisible by 2. You will have already determined it’s NOT prime when you tested P%2. So you can test for P%2 separately, and then start the loop with d=3 and do “d = d+2” to stick to just the odd numbers.

# Sieve of Eratosthenes

The principle behind the “Sieve of Eratosthenes” is to extend that last idea even further.

For example, there is no need to test if P can be divided by the multiples of 3 – 6, 9, 12, etc. And there’s no need to test the multiples of 5 – 10, 15, 20, etc., or 14, 21, 28, …, or 18, 27, 36, or ….

The Sieve is not calculating true/false for a single number P. It is calculating true/false for *all* the numbers P from 2 up to a maximum value N.

Create an array of booleans and initialize the array with A[i] = true for all i. Next loop over the array to set A[i] = false for all the multiples of 2 (except 2 itself, because 2 is prime). In other words, 2 is “in”, but all the multiples of 2 are “out”.

Then do this again for the multiples of 3, and then the multiples of 5, and 7, and so on.

When you’ve finished, the only A[i] that are still true will be the ones where i is prime.

You can also do a “square root” optimization with this algorithm, similar to what you could do with the brute-force algorithm above.

# The actual assignment

Implement both of the above algorithms and fill in as much of the table below as you can manage.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **#Primes <N** | **Calculation time in ms (Brute force)** | **Calculation time in ms (Eratosthenes)** |
| 10 | 4 | 0 | 0 |
| 33 | 11 | 0 | 0 |
| 100 | 25 | 0 | 0 |
| 333 | 67 | 0 | 0 |
| 1000 | 168 | 0 | 0 |
| 3333 | 470 | 1 | 0 |
| 10000 | 1229 | 2 | 1 |
| 33333 | 3569 |  |  |
| 100000 | 9592 |  |  |
| 333333 |  |  |  |
| 1000000 |  |  |  |
| 3333333 |  |  |  |
| 10000000 |  |  |  |
| 33333333 |  |  |  |
| 100000000 |  |  |  |
| 333333333 |  |  |  |
| 1000000000 |  |  |  |
| (Can you go any further?) |  |  |  |

Note: You do *not* need to write a program that produces all of the above output in one “go”. Rather, just make it easy to change the value of N in your program, and run it many times with different values.

For the Brute Force algorithm, write a function that takes an integer P and returns boolean indicating whether P is prime. Then in main(), call this function in a loop (2 to N) and count the number of “true” answers.

For the Eratosthenes algorithm, write a function that returns an *array of booleans*, A[i] = true if and only if i is prime. Then in main(), write a loop that counts the number of “true” elements in the returned array.

**How far should you go in the table?** As far as you “reasonably” can. The time your code requires will depend on (obviously) the size of the input N, the speed of your system, the efficiency of your code, which optimizations you implement, and of course how patient you are in waiting for your output. It will probably even vary a little bit just from one execution to the next because of variations in system resource management.

For the sake of comparison, on my laptop, which I think is reasonably speedy as laptops go these days, I was able to do N=10^7 by brute force and N=10^9 by Eratosthenes, without waiting too long (noticeable numbers of seconds, but well under a minute).

(Yep, I wrote a function that returned an array of a *billion booleans*. I should have named it *bullion*.)

# Tips and suggestions

You can verify (some of) your answers in the nifty table of results in Section 1.2 of this page:

<https://primes.utm.edu/howmany.html>

Time your Java code with System.currentTimeMillis(). This function returns the system time as a long integer. Call it immediately before and after some code, then subtract the two results to get the number of milliseconds your code took to execute.

There’s an animation of the Eratosthenes sieve algorithm on Wikipedia (and probably others out there, too):

<https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes>

Btw, that page also mentions a “starting from prime’s square” optimization for this algorithm. You don’t need to implement this optimization (or any others that I have not mentioned), but it’s okay if you do.

# Submission and marking

**Due date**: As shown on Learning Hub. Your last submission will be the one that counts. Late assignments will not be graded.

**What to submit**:

* Your Java source code (file name not important)
* The completed table, as far as you could get it (Word doc, Excel, plain text – whatever makes you happy)

**Marking**: This lab is worth 15 marks.

* Function implementing Brute Force “is prime” algorithm – 4 marks
* Function implementing Eratosthenes sieve algorithm – 4 marks
* Filling out the table of results – 4 marks
* Main program and coding style – 3 marks