Introduction

For this assignment, implement the Sieve of Eratosthenes using a pool of processes in Python, use test runs to see how quickly the program runs under different combinations of parameters, and write a short description of your results along with possible explanations for why the results came out that way.

The Sieve of Eratosthenes

The sieve is a simple way to find prime numbers that involves taking a list of integers and throwing away ones that are multiples. We start with an array `sieve` with `sieve[0]` and `sieve[1]` false and `sieve[2..`*n*`−1]` all true (where *n* is the length of the sieve). For each *k* = 2, 3, …, *n*/2, set `sieve[`*m***k*`]` false for all *m* ≥ 2 (and *m***k* < *n*, of course). When you're done, `sieve[`*p*`]` is true iff *p* is prime. Calculate a list of those *p*; that's the list of primes < *n*.
For the python version of the program, write a python3 program that declares a sieve array as a global variable and initializes it using `multiprocessing.sharedctypes.RawArray` (see the discussion below). The `main`() routine should run the sieve algorithm, return a list of prime numbers, and print out the time it took to calculate the sieve, in ms.

The main() routine should have keyword arguments `poolsize` (for the pool size to use) and `chunksize` (see below). To explain the chunk size, imagine a do_sieve(`k`) routine that removes all multiples of `k` from the sieve (but if `k` ≤ 1, it just removes `k`). To process the sieve, we'd need to run do_sieve on `k` = 0, 1, …, size of sieve - 1. Now imagine extending this routine to take `k` and `chunksize` and having it remove all multiples of `k`, `k+1`, `k+2`, …, `k+chunksize−1`. To process the entire sieve, we will run this routine with starting `k` values of `0`, `chunksize`, `2*chunksize`, and so on. These runs will be in parallel. A `chunksize` of 1 creates as many processes as possible to process the sieve; a
chunksize equal to the sieve size is a sequentially-running sieve.

Using Shared Ctype Data

The matrix multiplication routine from lecture doesn't have to share the results of the parallel computations between different processes. The top-level routine runs the parallel routines and collects the results.

For the sieve algorithm, we do have to share the sieve array among all the parallel sieving calls, since the final sieve array is the result of striking out all multiples of 2 and all multiples of 3 and so on. For the processes to share the sieve array, it must be global, as in the matrix multiplication program, but the we need `multiprocessing.sharedctypes.RawArray` to create the array. If you read the Python documentation on this class (and obviously, you should), you’ll find that the values produced by this class are shared across processes. The smallest unit of array data you can use are signed and unsigned characters. You’ll need to pick one of these types and decide on values you want to use to represent true and false.