Idea

1. search for all primes up to $\sqrt{n}$
   - reason (trivial): $a * b > n \; (if \quad a > b \; \vee \; b > a)$ for $a, b, c, \in \mathbb{N}$
   - uses *regular* [Sieve of Erathostenes]()
     $\longrightarrow$ represented as *array* of 1 and 0
2. split array up into *segments*
   - instead of one giant *numbers* array, the array is only created with the length of a segment
   - orientate towards CPU *L1d cache* (`lscpu | grep "cache"`)
     $\longrightarrow$ maximum efficiency
3. for each segment:
   - check for multiples of *all* primes found in *1)*
4. in all of this: *Ignore* all even numbers
   - even numbers cannot be prime
   - allows for only checking for every *other* multiple of a prime p because:
     - for every $p \neq 2 \in \mathbb{P}$ is true: $2 \nmid p$
     - $2 \nmid p \rightarrow 2 \nmid (2a+1)p \wedge 2 \mid 2a * p \; (a \in \mathbb{N}_0)$
     $\longrightarrow$ since $2p$ is obviously *even*, it gets ignored by *every* part of the program, thus its state is irrelevant
     $\Longrightarrow$ this means that when counting primes, *2 is never included* $\longrightarrow$ thus, the number of found primes is always *one less* than the actual number of primes

```
is_prime[] = regular_sieve(until sqrt(n))
num_primes = 1 # compensates for ignoring 2
for each segment [low, high]:
        sieving_primes = [primes in is_prime until sqrt(high)]
        multiples: list[len(sieving_primes)]

        for each prime p in sieving_primes:
                a = multiples[index of p] # a is always uneven
                for each multiple m = 2a * p:
                        eliminate m from segment
                when multiple > high:
                        store m in multiples

        num_primes += number of primes in segment
```

Why it is faster

Locality of Memory

A regular sieve is *massively* ineffient when it comes to caching, as each time, the processor has to go over the *entire* array. Splitting it up into smaller chunks easens the workload of the processor, and also, after finishing each segment, all non-primes disappear fom the cache. Making it for once more space-efficient, but also more time-efficient due to *locality of memory* - whatever that is exactly.

Ignoring even numbers

Only dealing with uneven numbers essentially *halves* the workload. This can also be observed in the time: When processing *all* numbers, runtime jumps from *~250 ms* all the

Implementation:

[Implementation Segmented Sieve]