Implementation Segmented Sieve

Code

segmented_sieve.c

Runtime

Time Complexity: $O(n\ log(n))$
Space Complexity: $O(\sqrt{n})$

Timed (average-case):
as *linux executable*: ~225 ms
as *windows executable* under `wine`: ~260 ms

Comparison

This compares the *average case* approximate runtime of the following entities:

- Segmented Sieve (with hardcoded values): segmented_sieve_hardcoded.c
- Segmented Sieve: segmented_sieve.c
  - under `wine`: Compiled to `exe`, then ran via the wine linux library. This serves to better compare the Benchmark, which is provided as a windows executable.
- Benchmark: https://elearning.dhbw-stuttgart.de/moodle/pluginfile.php/660507/mod_folder/content/0/PrimDynC.exe?forcedownload=1 --> *PrimDynC.exe*
  - This is the program that also allows for User Input. *PrimOpt2*, which does not do so, is faster, but is hardcoded to 10^8.

10^8

	Segmented Sieve (with hardcoded values)	**Segmented Sieve**	**Segmented Sieve (under `wine`)**	**Benchmark (under `wine`)**	**Benchmark (with hardcoded values)**
battery power	230 ms	280 ms	340 ms	850 ms	700 ms
performance mode	225 ms	?? ms	?? ms	?? ms	340 ms

--> As *space complexity* is optimized, Segmented Sieve runs basically equally fast regardless of available resources
--> `wine` emulator adds 20% of runtime

10^9

	Segmented Sieve (with hardcoded values)	Segmented Sieve	Segmented Sieve (under `wine`)	Benchmark (under `wine`)
battery power	2450 ms	2900 ms	3300 ms	6900 ms

10^{10}

Sadly, under `wine`, both programs don't run properly anymore, as theyre lacking storage space. This is a shame.

The regular segmented_sieve.c needs ~ 35 seconds.The regular segmented_sieve.c needs ~35 seconds.

Documentation

> since I need `stdio` for this and it redefines `timeval` from `linux/time.h`, I'm not working with the provided Nimmzeit module, instead using `clock()` to measure the time. A giant Fuck You to hoever wrote that stupid module.
>
> Futhermore, since the evaluation happens *inside* the sieving, it cannot be timed seperately. Of course one could time *each* individual sub-evaluation, however that slows the program down by ~50 ms, which is about 20% of total runtime and thus not feasible.

Storage of Sieving primes

Structure list_uint64

Roughly emulates a *list* in python.

```
typedef struct
{
    uint64_t *data;
    uint64_t max_size;
    uint64_t length;
} list_uint64;
```

Attributes

- *data* -> Pointer to where the *actual data* of the list lies. This can be used synonimously with an array using bracket syntax: `list.data[index]`.
- *max_size* -> How much *memory* has been allocated. Is used when creating the `multiples` list, to keep its size in sync with the `primes` list.

- *length* -> equivalent to the return value of `len(list)` in python. Stores how many elements the list currently holds. Can be used to iterate over only the parts of the list that actually store values.

Initialization with default values

Parameter size: The maximum size the List should be.
Return value: The newly created list. Since structs don't need explicit pointers to be passed around, the list instance can simply be returned.

```
// creates a new list filled with default values
list_uint64 new_list(uint64_t size)
{
    list_uint64 list = {
        malloc(size * sizeof(uint64_t)),
        size,
        0
    };

    return list;
}
```

- *data*: Manually allocates $size * sizeof(uint64\_t)$ of memory. This way there can be *size* Elements of the type *uint64_t* stored in the list.
- *max_length*: This is initialized to the given *size* of the wanted list.
- *length*: Trivial - As there are no initial elements in the list, its size should be 0.

Appending

```
// appends an item to given list. List is passed as reference.
void list_append(list_uint64 *list, uint64_t value) {
    list->data[list->length++] = value;
}
```

This is more or less a very fancy *array syntax*. It first accesses the data list via pointer operations (`list->data`), then stores the value at the end (`list->length`) and then increases the length attribute by one.

Setup

1. create `is_prime` array to later run a regular sieve on
2. fill this array with one's --> leave indexes for 0 and 1 out, because those aren't primes
3. run a regular Sieve of Erathostenes > C implementation on the `is_prime` array
   --> places all indexes of primes until $\sqrt{n}$ with 1, else 0
   --> using `is_prime[index]` will return whether *index* is *prime*

4. create a `list_uint64` instance to later store the sieving primes in
    - The size of the array is approximated: Approximation of Primes

```
bool is_prime[(size_t)sqrt(upper_limit)];
memset(is_prime + 2, true, sizeof(is_prime) - 2);
sieve(is_prime, sizeof(is_prime));

list_uint64 primes = new_list(PRIMES_UNTIL_SQRT);
```

Approximation of Primes

> The prime number theorem suggests that $\lim_{x\to\infty} x\ log_{10}(x) = \pi(x)$

Since $x\ log(x)$ generally falls short of the actual number of primes, however we want to rather *overshoot* than *underestimate*, The result has to be increased. by some amount.

Let $A_x = \frac{\pi(x)}{x \log_{10}(x)}$

x	$\pi(x)$	A_x
10	4	0.921
10^2	25	1.151
10^3	168	1.161
10^4	1229	1.132
10^5	9592	1.104
10^6	78498	1.084
10^7	664579	1.071
10^8	5761455	1.061
10^9	50847534	1.054
10^{10}	455052511	1.048

In this table can be seen, that $\forall_x A_x \leq 1.52$. So, to get a number a bit larger than the actual number of primes until a parameter *n*, this code can be used:

```
int n;
(n / (uint64_t)log10(n)) * 1.2
```

Algorithm

1. Initialization of relevant variables

Scope of function

> Variables that will be used over and over again and incremented in each segment

```
uint64_t low, high;
uint64_t num_primes = 1;
uint64_t sieving_prime_candidate = 3;
uint64_t prime_counter = 3;
list_uint64 multiples = new_list(sieving_primes.max_size);

bool segment[segment_size];
```

- *low, high*: control boundaries of each segment
- *num_primes*: serves as counter for the total number of primes
  - each segment adds the number of primes it contains to this variable
  - variable is initialized to *1* because the entire code ignores multiples of 2
- *sieving_prime_candidate*: bit of a chunky name, but tells exactly what the variable is for
  - value will increase with each segment
  - this variable will get checked against `is_prime` to see if its value is prime
- *prime_counter*: used at the end of each segment to iterate through the `segment` array to find all primes
  - not to be confused with *num_primes*
- *multiples*: stores the first multiple of a prime that is *outside* the current segment, so the next segment will know where to start eliminating for the prime
  - since there is exactly *one* "next multiple" for every sieving prime, the lists have the same size, and `multiples.data[i]` will always correspond to `sieving_primes.data[i]`
- *segment*: the good old essential Sieve of Eratosthenes array for a given segment.
  - will be reused for different segments
    --> data of one segment *does not persist*
  - attention: since the entire program *ignores even numbers*, every even index will also be marked as *true*, just not evaluated.

Scope of outer loop

> Sets environment for the current segment.

```
for (low = 0; low <= upper_limit; low += segment_size)
    {
        memset(segment, true, sizeof(segment));
        high = low + segment_size - 1;
        if (high > upper_limit)
            high = upper_limit;
```

- Loop condition is trivial

- sets *every* element of the `segment` array to *true*
- high is either set to *one less* than the low of the next segment or the upper limit (trivial)

2. Filtering of Sieving Primes

> *Sieving Prime*: A prime p that is used to cross of multiples $a * p$ within a segment $[l, n]$. Sieving Primes only need to be smaller than $\sqrt{n}$.

```
for (; sieving_prime_canidate * sieving_prime_canidate <= high;
sieving_prime_canidate += 2)
{
        if (is_prime[sieving_prime_canidate])
        {
                list_append(&sieving_primes, sieving_prime_canidate);
                list_append(
                        &multiples,
                        (sieving_prime_canidate * sieving_prime_canidate)
- low
                );
        }
}
```

- sieving prime candidates always get increased by 2
  --> since `sieving_prime_candidate` is initialized to 3, this ignores *all even numbers*
- If a candidate is in fact a prime, it is appended to the *list* `sieving_primes`
  - also, a corresponding entry is appended to the `multiples` list to be able to be used *later on*
  - The default value for the new `multiples` entry can be derived from the common standard sieve optimization, to start eliminating from $p * p$ instead of p

3. Eliminating multiples

```
for (uint64_t i = 0; i < sieving_primes.length; i++)
{
        uint64_t j = multiples.data[i];
        for (; j < segment_size; j += sieving_primes.data[i] * 2)
        {
                segment[j] = false;
        }
        multiples.data[i] = j - segment_size;
}
```

- Iterator variable j is assigned the corresponding `multiples` entry
  --> if the sieving prime p has been used in the segment before, this will be the first multiple of p in the current segment

- Since all even indexes get ignored, we can only check every *other* multiple of p
  - $2 \nmid p \rightarrow 2 \nmid (2a+1)p \wedge 2 \mid 2a * p$, therefore $2a * p \notin \mathbb{P}$
- Finally, the `multiples` entry is updated to store the *next* multiple, that doesn't lie in the current segment
  - this usage of j is why j needed to be declared *before* the for loop

4. Evaluation

> Since data of an individual segment doesn't persist, the evaluation needs to take place inside of a segment.

```
for (; prime_counter <= high; prime_counter += 2)
{
        if (segment[prime_counter - low])
        {
                num_primes++;
        }
}
```

- Works the same way as the evaluation of a regular Sieve of Eratosthenes.

Idea

> Segmented Sieve
>
> 1. search for all primes up to $\sqrt{n}$
>   - reason (trivial): $a * b > n \ (if \quad a > \sqrt{n} \ \wedge \ b > \sqrt{n})$ for $a, b, c, \in \mathbb{N}$
>   - uses *regular* Sieve of Erathostenes
>   --> 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
> 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: $2 \nmid p$
>   - $2 \nmid p \rightarrow 2 \nmid (2a+1)p \wedge 2 \mid 2a * p \ (a \in \mathbb{N}_0)$
>   --> since $2a * p$ is obviously *even*, it gets ignored by *every* part of the program, thus its state is irrelevant

> ==> this means that when counting primes, *2 is never included* --> thus, the number of found primes is always *one less* than the actual number of primes

pseudo code

```
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 way to *~500 ms*

Implementation

Implementation Segmented Sieve

Reference

explanation of algorithm: https://github.com/kimwalisch/primesieve/wiki/Segmented-sieve-of-Eratosthenes