Homework 1 - Sieve of Eratosthenes

by Kaloyan Doychinov, COS2021A

2. Spend some time with pencil and paper attempting to approximate the time and space complexity required by this algorithm. Don't write any code, yet

So, what I have done in this part of the assignment is writing some ideas and pseudo code in the "pseudo.txt" file provided (you can also check it below).

Summarized Results:

> Space Complexity: n
> Time Complexity: 1/2 * n * log(n)

Here it ("thought process" - I kind of think like code - so I immediately opened vim and started writing something) is too:

```
// starting at 2 since 0/1 not primes, and getting input
i = 2; n = input

// just as the text's said - we need a table to cross out values
table[n] = true
table[0..1] = false

// we loop to sqrt(n) as said in the example
while i < sqrt(n):
    // we skip (as the txt says) the non-primes
    if(!table[i]) continue

    // if we find a prime - we cross all of its prime*1,2,3..n/prime
    // why n/prime you may ask - because otherwise we would be overflowing the table
    for j = 2; i*j < n; ++j
        table[i*j] = false // and just set them all to false
    ++i
printing...
end

other things:
1, 2 - cuz its not
3, 4, 5, 6, 7, 8, 9, 10

time complexity:
    outer loop -> sqrt(n)
    inner loop (only if not cross-out) -> n/i
    overall: 1/2*n*log(n)
space complexity: just n, you could optimize it to n-2
```

```
but does it really matter (and it could technically be made
down to n/2 - if we exclude even numbers as they are always non-prime)

more detes for time complexity:
---------inner--loop----------
0 ------------------------ n/2
- ========================
- =================
- ============
- ======
- ===
- =
--------------------------------
-> n/i - complexity-wise for the inner loop
(looks like a logarithm but I'm no sure)
=> to calculate the time complexity of the algorithm,
we would need the outerloop's context

so it basically looks like a SUM(2..sqrt(n)){n/i},
since n is constant we can pull it out
=> n*SUM(2..sqrt(n)){1/i} since i runs to sqrt(n),
since this is where my math runs to - I would approximate it's
~n*log(sqrt(n)) which is like 1/2*n*log(n)

=> sqrt(n) * n/i
=> i is 2 .. sqrt(n) => we can create a time series log(sqrt(n))
=> time complexity is 1/2*n*log(n)
=> BigO = n*log(n)
```

3. Once you have an approximation (don't worry if you suspect it's incorrect), implement Q:2.21 in C++. The program should take a single command line argument for N. For example, if your executable is called sieve, I should be able to run, sieve 100, and it will find all primes between 2 and 100 inclusive. Verify the solution is correct by using your sieve to find and print all prime numbers less than 100

Here's a quick preview of the algorithm of my C++ code (the whole can be find in `sieve.cpp`):

```cpp
for(int i = 2; i <= sqrt(n); ++i) {
    if(!table[i]) continue;

    for(int j = 2; i*j < n; ++j) {
        table[i*j] = false;
    }
}
```

You can compile it using g++ or clang, and run it (with a mandatory argument N - positive integer):

```
g++ sieve.cpp -o sieve
./sieve 100
```

4. Make two copies of your implementation called timing_sieve.cpp and counter_sieve.cpp . . .

For this section, I will provide a table with the time complexity and the counter and timings between different various Ns:

N	Time (ns)	Count	C to N	T to N	Calc from 2.
100	1375	116	1.160	13.75	100
500	5542	728	1.456	11.084	674
1000	11667	1566	1.566	11.667	1500
5000	60041	8665	1.733	12.008	9247
10000	123333	18063	1.806	12.333	20000
100000	1753417	202402	2.024	17,534	250000
1000000	10695791	2198668	2.199	10,696	3000000

- C to N - Count compared to Ns
- C to T - Time (ns) compared to Ns
- Calc from 2. - Calculations for Expected *Count* from 2.

5. Answer the following question

Does the complexity of your implementation match your initial pencil and paper estimate?

Speaking of ***Counter***:

NO????? -> that was my first answer because the numbers were wayyyyy higher - but then I noticed that I had a mistake in my calculations => fixed them to 1/2*n*log(n) aka bigO = nlog(n) - and all the numbers aligned ~~

Speaking of the correct numbers of the counter - they're approximately correct. When the values are small (until ~1000) it's over the expected count. Just like we've learned at class. (deducted from the ration "C to N")

If not, why do you believe this is the case? We will discuss this in class after the assignment is handed in

Speaking of ***Timing***:

My guess (as I do love computers since I was a kid - and I especially loved (still do) watching YouTube videos) - is the CPU's branch prediction of the calculations in addition with its cache - therefore it remains relatively the same

relative to N performance - even though it iterates really big N times more. (deducted from the ratio "T to N" which is relatively the same with a few MoE deviations)

6. Document your experience in a short write-up...

Overall, when we take into account both theoretical (counting) and actual (time) complexity - we can see how our code is not really ours - we don't fully control it (unless we write Assembly). As I have mentioned above we have factors like CPU branch predictions, CPU cache, etc. But also the modern-day compilers are so optimized (gcc, clang, etc.) that they change the structure of the code in order to make it more memory efficient and with higher performance. That's why C/C++ are high level languages - the level of abstraction from what the actual machine does, is quite big.

And just another note about the compilers and their optimizations - e.g. I was almost going to give you non-working code because I have had forgotten to add `<cmath>` and `<vector>` - but clang on my mac - just included them under the hood, which is great in one way - BUT IT COULD HAVE ENDED REALLY BADLY. Thankfully I have done this mistake before and checked if it compiled and ran on GNU/Linux.

Thank you for reading this Homework Assignment #1

Additional Notes

Silvia and I worked together, even tough it may not seem that way - we helped each other in different aspects - particularly brainstorming or some small details