Prime numbers

Motivation. For increasing the table size in hashing, we wanted to find a prime of a certain size. Specifically, given an integer $M > 1$, we would like to find a prime of size at least $2M$.

Our approach is based on *Bertrand's postulate*:

For all $n > 1$, there is a prime p such that

$$n < p < 2n.$$

Q1. Give an $O(M\sqrt{M})$ algorithm to find the next prime that is at least $2M$ (solution below).

We can iterate over $2M \leq x \leq 4M$, and break as soon as we find that x is prime.

The most naive version of `is_prime(n)` is to test whether n is divisible by each of $2, 3, \ldots, n-1$, in $O(n)$ time.

One improvement we can make is to only test whether n is divisible by a divisor in $2, \ldots, \lfloor\sqrt{n}\rfloor$. If n has a factor between 2 and $n-1$, then one of them has to be at most $\sqrt{n}$.

An optimization is to only test if n is divisible by *odd numbers* at most $\sqrt{n}$ (and check two separately). This improvement gives an algorithm that only does half the checks and is about twice as fast, but has the same complexity asymptotically.

A key observation is that it is sufficient to only iterate over *prime numbers* at most $\sqrt{n}$. If x divides n, then every prime divisor of x divides n. This should significantly improve `is_prime(n)`: for intuition, there are only 25 primes that are at most 100, 168 primes that are at most 1000, 1229 primes that are at most 10000. More concretely, if $\pi(n) :=$ number of primes in $[1, n]$, then

$$\pi(n) \in \Theta\left(\frac{n}{\log n}\right).$$

For completeness we mention that if $n \geq 55$,

$$\frac{n}{\ln n + 2} < \pi(n) < \frac{n}{\ln n - 4}.$$

So if we know the primes that are at most $\sqrt{n}$, we can check if n is prime in $O(\sqrt{n}/\log n)$.

Sieve of Eratosthenes. This is a very well-known and practical algorithm for computing primes in $[0..n]$ in $O(n \log\log n)$ time. The idea is to write down all numbers between 2 and n, and initialize `is_prime[x]` to true for all of them, and to iterate over them in increasing order. Every time we arrive at a number x that has not been "crossed out" (i.e. `is_prime[x]` is true), we "cross out" all the multiples of x starting with $x \cdot x$.

```
is_prime[0..n] = [true, ..., true]
is_prime[0] = is_prime[1] = false
for i = 2..n:
    if is_prime[i]:
        for j = i*i..n, j += n: // i*2, i*3, ... already crossed
            out
            is_prime[j] = false
```

Q2. Argue the runtime of $O(n \log n)$.

Q3. Our next goal is to prove the runtime is $\Theta(n \log\log n)$.

(a) Denote by p_n the n-th prime number. Show that $p_n \in \Theta(n \log n)$ using the asymptotic bound on $\pi(n)$.

(b) Give a proof sketch that the runtime is in $O(n \log\log n)$ using bounds with integration.

The lower bound is obtained from exactly the same argument as in part (b).

A few technical notes about implementation:

- to obtain *just* the primes, once we know i is prime, we can push-back i into a dynamic array
- we should be careful computing `i*i` because it might overflow
- depending on the use case (for example if we are *not* storing the primes as in the first point), we may even stop as soon as `i*i > n`

  This optimization does not improve the runtime asymptotically (as seen in Q2), but it is noticably faster when implemented.

Q4. Give a way to reduce space and the number of operations performed by the Sieve approximately in half.

Q5. Give an $O(n \log n \log\log n)$ algorithm to count the number of factors in the prime factorization for each of $2, \ldots, n$.

For example, $20 = 2 \cdot 2 \cdot 5$, so 20 has 3 factors in its factorization. Similarly, $35 = 5 \cdot 7$ has 2 factors in its factorization.

Q6. How can the Sieve of Eratosthenes be adapted to compute all prime numbers in a particular range $[a, b]$ when b is far too big to compute all primes in $[0, b]$?

Hint: the intended complexity is $O(\sqrt{b}\log\log b + (b-a)\log\log b)$.

While hash tables of such big size are unlikely to occur in practice, this variant is perhaps more natural when considering table doubling.

Number theoretic functions. We first discuss several well-known functions from number theory.[1]

For the next problems, assume that $n = p_1^{a_1} \cdots p_k^{a_k}$ where $p_1, \ldots, p_k$ are the first k primes and all $a_i \geq 0$. We also assume we already computed the 1-indexed array `primes` with the first (sufficiently many) primes.

The *number of* ***prime*** *factors* of n, which is $\sum a_i$, is easy to find. We have computed it from scratch in Q5.

Q7. Give a way to compute the number of prime factors of n (assuming we already have the array `primes`).

Hint: intended solution has time $\Theta(\pi(\sqrt{n}))$.

Q8. Using Q7, implement `factor`, which given an integer n returns a dynamic array containing all the prime factors of n. The time complexity should be the same.

For example, if $n = 12$, then `factor(n)` returns `[2, 2, 3]`.

We should note that the worst case for the algorithm in Q8 is very slow. If n is a prime, then the algorithm tests all the first $\pi(\sqrt{n})$ primes. Although there are optimizations to the algorithm in Q8, we still do not have a computationally feasible way to factor large numbers, which serves as the basis for cryptography[2] It is fun to think about some special cases nonetheless.

Fermat's factorization method. We can write an odd composite number $n = p \cdot q$ as a difference of squares:

$$n = \underbrace{\left(\frac{p+q}{2}\right)^2}_{a^2} - \underbrace{\left(\frac{p-q}{2}\right)^2}_{b^2}.$$

This factorization method guesses the first square a^2 (as $\sqrt{n}$, the smallest it can be), and then checks if $b^2 = a^2 - n$ is *also* a square. If it is return, otherwise increase a and try again.

[1]Perhaps you are already familiar with many of these from before. Either way, if you are interested in learning more, one fun class to take is PMATH 341. Keywords: multiplicative functions.

[2]See, for example, CO487.

```
fermat(n):
    // returns a factor of n
    a = ceil(sqrt(n))

    b_squared = a*a - n
    b = round(sqrt(b_squared))

    while b * b != b_squared: // while b is not a perfect square
        ++a // try the next guess
        b_squared = a*a - n
        b = round(sqrt(b_squared))
```

Q9. Give the asymptotic runtime of this algorithm. When does this algorithm perform well?

It is interesting to note that implementations of encryption that relies on difficulty of factoring are made extremely vulnerable if consecutive primes are used in key generation because of this Factorization method.

We denote by $\tau(n)$ the *number of factors* of n. For example, $n = 60$ has four prime factors: 2, 2, 3, 5. Given the prime factorization of n we may compute it as, $\tau(n) = \prod(a_i + 1)$.

Q10. Give a way to count the number of divisors of n given only the array `prime`.