1 Checking Primes:

When you think of checking primes, what is the most basic algorithm that can come to mind? This relies on what it means to be prime: the only numbers that divide the number of interest, $n$, must be $n$ and 1. Note that we will define 1 to not be prime.

We can then consider this basic algorithm, which will loop from $i = 2$ to $i = n$, checking all possible integers between them until for some $i$, $n \mod i = 0$ (in which case $n$ is not prime), or whether for all $i$, $n \mod i \neq 0$ (in which case $n$ is prime).

In code, this can look like the following:

```scheme
(define (prime? n)
  (letrec ((loop (lambda (i)
                   (cond ((>= i n) #t)
                         ((divisible? n i) #f)
                         (else (loop (+ i 1)))))))
    (loop 2)))
```

1.1 What is the space complexity of this algorithm?

Since we don’t need to store any items, the space required should be $O(1)$.

1.2 Now what about the time complexity of this algorithm?

We have to check `divisible?` at each $i$ from $i \in [2, n)$. For this problem, let’s assume that modulo operates in constant time.\(^1\) So we get a time complexity of $c + \ldots + c$, which is in $O(n)$.\(^n-2\) times

2 Can We Do Better?

Now that we have established a baseline for checking if a number is prime, we can use some facts about division and multiplication to our advantage. Note that if a number $n$ is divisible by $p$, then it is also divisible by $n/p$. The way our current algorithm works is that it checks both of these factors, when optimally it would only have to check one. The main question is, what is the smallest value of $p$ to check to guarantee that we have enumerated all possible pairs of factors?

\(^1\)Note that with larger integers, modulo relies on the number of bits in the integer. For the purpose of this handout, we will conduct a simplified analysis assuming that modulo is constant.
Hopefully after some thinking, we should agree upon on $p = \sqrt{n}$. This is because then $p = n/p = \sqrt{n}$, and checking anything above $\sqrt{n}$ guarantees that the other integer in the pair has been checked. Why is this?

Proof: Consider otherwise. Then we will have enumerated all $p < \sqrt{n}$ but there exists some $q > \sqrt{n}$ such that $n/q > \sqrt{n}$ (otherwise it would have been enumerated already). But then $n > q\sqrt{n} > n$, which is impossible. Hence we have derived a contradiction.

So let’s modify the algorithm:

```scheme
(define (prime? n)
  (letrec ((loop (lambda (i)
                   (cond ((> i (sqrt n)) #t)
                         ((divisible? n i) #f)
                         (else (loop (+ i 1)))))))
    (loop 2)))
```

2.1 **What is the space complexity of this algorithm?**

Since we still don’t need to store any items, the space required should remain $O(1)$.

2.2 **Now what about the time complexity of this algorithm?**

We have to check `divisible?` at each $i$ from $i \in [2, \sqrt{n}]$. For this problem, let’s assume that modulo operates in constant time. So we get a time complexity of $c + \ldots + c$, which is in $O(\sqrt{n})$.

3 **Sieve**

By now we have seen sieve in class:

```scheme
(define sieve
  (lambda ((stream <stream>))
    (stream-cons
      (stream-first stream)
      (sieve (filters (lambda ((x <integer>))
                           (not (divisible? x (stream-first stream))))
              (stream-rest stream))))))

(define primes
  (sieve (stream-rest integers)))
```

Note that filters is delayed: this means the stream isn’t instantly filtered, but will filter at every step. So each check of divisibility does not occur until we move to the next element of integers. As we get more primes, we have to check the next number against more divisors. We discussed in class that we can bound the amount of primes under $n$ by $\log n$ (for more information, see the prime-counting function). Hence at most we will have to check by at most $\log n$ numbers at the last step.

3.1 **What is the space complexity of this algorithm?**

We need to return a stream of primes, and need to scan $n$ integers. This is $\log n$ to return primes, which is $O(\log n)$. 
3.2 Now what about the time complexity of this algorithm?

We have to check at worst case \( \log n \) elements each time.\(^2\) So we can model this as the sum

\[
\sum_{i=2}^{n} \log i.
\]

We know that \( \log n \) is a monotonically increasing function (this means that as if \( m > n \), then \( \log m > \log n \)). So we can treat this sum as a left rectangular approximation to

\[
\int_{2}^{n+1} \log n,
\]

and we know that

\[
\sum_{i=2}^{n} \log i < \int_{2}^{n+1} \log i \, di,
\]

since \( \log n \) is monotonically increasing. This integral evaluates to \( (n + 1) \log(n + 1) - (n + 1)/\ln 10 - 2\log 2 - 2/\ln 10 \). Note that \( \log(n + 1) = \log(n + 1/\sqrt{n}) = \log(n) + \log(\sqrt{n}) \). Here, \( \log(n + 1/\sqrt{n}) < 1 \) when \( n \geq 2 \).

So hence, this algorithm runs in \( O(n \log n) \).

4 Comparing Approaches

So what happens if we want to generate all primes under some number \( n \)?

We’ve seen how Sieve does it, but what about looping through integers 2 through \( n \) and keeping only the prime integers. Something like:

```scheme
(define (primes n)
  (letrec ((loop
                (lambda (x accum)
                  (cond ((> x n) accum)
                        ((prime? x) (loop (+ x 1) (cons x accum)))
                         (else (loop (+ x 1) accum))))))
    (loop 2 '())))
```

The space complexity would be \( O(\log n) \) to make the list.

What about the time complexity? Note that \( \sqrt{x} \) is also a monotonically increasing function. So we can use the same ”integral trick” to bound the sum

\[
\sum_{i=2}^{n} \sqrt{n}
\]

by above. We will calculate

\[
\int_{2}^{n+1} \sqrt{n} = \frac{2}{3} \left( (n + 1)^{3/2} - 2^{3/2} \right) = \frac{2}{3} \left( n\sqrt{n + 1} + \sqrt{n + 1} - 2^{3/2} \right).
\]

\(^2\)Note that usually this won’t happen, since not many numbers are prime and we can filter out the element earlier than expected, but for this assignment we will provide a simpler model at the cost of a looser upper bound.
By the same logic as above, we have that $\sqrt{n + 1} = \sqrt{n \frac{n+1}{n}} = \sqrt{n} \sqrt{\frac{n+1}{n}}$. Note that for $n \geq 2$, $\frac{n+1}{n} < 2$. Hence we have that this algorithm is in $O(n \sqrt{n})$.

What does this mean? Well, for calculating all primes under $n$, we can use Sieve, which is better since $O(n \log n)$ is faster than $O(n \sqrt{n})$. But for checking if one number is prime, it is better to use prime, since you need not generate all primes before it.

5 Appendix: Miscellaneous Scheme Functions

(define (divisible? n m)
  (zero? (modulo n m)))

(define ones
  (stream-cons 1 ones))

(define (add-streams s1 s2)
  (stream-cons (+ (stream-first s1)
                  (stream-first s2))
               (add-streams (stream-rest s1)
                          (stream-rest s2)))))

(define integers
  (stream-cons 1 (add-streams ones integers)))