Recitation 1

Introduction

1.1 Announcements

- Welcome to 15-210!
- The course website is `http://www.cs.cmu.edu/~15210/`. It contains the syllabus, schedule, library documentation, staff contact information, and other useful resources.
- We will be using Piazza (`https://piazza.com/`) as a hub for course announcements and general questions pertaining to the course. Please check it frequently to make sure you don't miss anything.
- The office hours schedule is posted on the course website as well as Piazza. Come meet all of your TAs!
- The first homework assignment, *IntegralLab*, has been released! It's due **Friday at 5pm**, but don't worry – it's quite short.
- Homeworks will be distributed through Autolab (`https://autolab.andrew.cmu.edu/`). Most homework assignments will be released on Fridays and will be due one week later. You will submit coding tasks on Autolab, and written tasks on Gradescope (`https://gradescope.com/`).

1.2 Sequences

Sequences, which you've seen in 15-150, are a parallel data structure that are immutable and functional like lists, yet easy to randomly access like arrays. They're a useful abstraction for various real-world parallel sequence structures that let us model parallel computation at the algorithmic level.

Several basic operations are useful on sequences, like tabulations, maps, filters, and reductions. We use mathematical notation to represent these manipulations:

$\langle e : 0 \leq i < n \rangle$	`tabulate (fn i => e) n`
$\langle e : p \in S \rangle$	`map (fn p => e) S`
$\langle p \in S \mid e \rangle$	`filter (fn p => e) S`

Using an efficient sequence, like `ArraySequence`, the work of `tabulate`, `map`, `filter`, and `reduce` is linear in the size of the sequence if the function argument is constant-time. However, we get large gains in span: constant for `tabulate` and `map`, and logarithmic for `filter` and `reduce`.

1.3 Primes

Let's now get some practice using parallel sequences.

Task 1.1. *Implement the function `isPrime : int → bool`, which returns `true` iff the input is a prime number.*

Task 1.2. *Determine the work and span of `isPrime` n.*

Now that we have a way of testing primality, we can try to generate a sequence of primes.

Task 1.3. *Implement the function `primes : int → int seq`, which returns a sequence of the prime numbers less than n.*

The work and span of this algorithm are more subtle. Using tools that we'll develop later in the course, we are able to derive a $O(n^{3/2})$ work and $O(\log n)$ span bound for the simplest prime generation algorithm.

1.4 Bonus Exercises

There's a more efficient way to generate primes up to n. Knowing a number is composite, we know that all multiples of it up to n are composite. So, for each number from 1 to $\sqrt{n}$, we can keep track of its multiples, and for each composite we can mark its multiples composite. This algorithm is called the sieve of Eratosthenes, and it is much more efficient.

Task 1.4. *Implement the function* `primes : int -> int seq` *using the improved method.*

.