Function: `repeating-stream`

Time Complexity: $O([n \to n])$, where n is the number of elements accessed from the stream with `take`.

Reasoning: The function `repeating-stream` first calculates the length of list `numbers`, an operation which is linear in the length of `numbers`, some constant. It then calls function `tail` exactly once. `tail` performs some arithmetic operations, accesses an element in `numbers`, then calls itself recursively—all of which happens in constant time. It recurs once for each element accessed from the stream with `take` (which is the value of n), because `tail` is required to generate the next value in the stream. Since `tail` takes constant time and runs n times, `repeating-stream` takes linear time.

Function: `fraction-stream`

Time Complexity: $O([n \to n^2])$, where n is the number of elements accessed from the stream with `take`.

Reasoning: For each element accessed with `take`, the function `fraction-stream` calls `fraction-stream-helper` once. `fraction-stream-helper` performs some simple constant time list manipulation, then calls `calculate-approx` once. `calculate-approx` does some arithmetic using a recursive function call. It recurs once for each element before the current element in the stream. Meaning, in the worst case, `calculate-approx` will recur n times—giving it a linear time complexity. Since `fraction-stream` runs n times, each time calling `calculate-approx`, which has a linear time complexity, `fraction-stream` has a quadratic time complexity.

Function: `threshold`

Time Complexity: $O([n \to n])$, where n is the number of elements accessed from the stream with `take`.

Reasoning: For each element accessed with `take`, `threshold` accesses it's first and rest fields, which are then used to perform some arithmetic—all of which occurs in constant time. It then checks if the result of the arithmetic is less than a given value, in which case it terminates, otherwise it recurs. `threshold` always runs n times, because it accesses terms one at a time until the condition is satisfied. Since `threshold` performs constant time computation n times it has a linear time complexity.

Function: `terminating-stream`

Time Complexity: $O([n \rightarrow n])$, where n is the number of elements accessed from the stream with `take`.

Reasoning: For each element accessed with `take`, `terminating-stream` deconstructs the list `numbers` either returning `nones` or a `lz-link` containing the next value in `numbers` along with a recursive function call. Although, `terminating-stream` terminates upon returning `nones`, `nones` is still a stream that will have to call a function each time a further element is accessed—this function runs in constant time, simply returning a `lz-link` of `none` and a recursive function call. If instead the `lz-link` is returned, `terminating-stream` will continue recuring until `numbers` runs out of elements, at which point `nones` will be returned. Since constant time computation will have to be executed for each element accessed with `take` no matter what, `terminating-stream` has a linear time complexity.

Function: `repeating-stream-opt`

Time Complexity: $O([n \rightarrow n])$, where n is the number of elements accessed from the stream with `take`.

Reasoning: `repeating-stream-opt` has identical code to `repeating-stream` except it wraps each value in the stream it returns in the `some` field of the `Option` type. This extra computation is very minimal and occurs in constant time, meaning the same logic from the reasoning for `repeating-stream` again applies.

Function: `fraction-stream-opt`

Time Complexity: $O([n \rightarrow n])$, where n is the number of elements accessed from the stream with `take`.

Reasoning: `fraction-stream-opt` has identical code to `fraction-stream` except for two differences. Firstly, since `coefficients` is now of the `Stream<Option<Number>>` type, the function must deconstruct each value from the `Option` type before use. The function does so in constant time with `cases` and then proceeds with processing it exactly the same as `fraction-stream`, even calling the exact same `calculate-approx` function. This extra computation is very minimal and occurs in constant time, meaning the same logic from the reasoning for `fraction-stream` again applies.

Function: `threshold-opt`

Time Complexity: $O([n \rightarrow n])$, where n is the number of elements accessed from the stream with `take`.

Reasoning: For each element accessed with `take`, `threshold-opt` first checks if the `approximations` stream has run out of terms (i.e. either of the next two terms is a `none`), at which point it raises an error, halting computation. If this is not the case (i.e. both the next two terms are a `some`), `threshold-opt` perform some constant time arithmetic. Next, depending on the result, the function either returns the value of the current term or recurs on the rest of the stream. Although `threshold-opt` can terminate early, in the worst-case scenario (where the final term accessed is the first to meet the threshold condition), `threshold-opt` recurs n times. Since in the worst case `threshold-opt` performs constant time computation n times it has a linear time complexity.