The following problems involve the `MutableIntCursor` ADT that represents a list of integers with the additional ability to insert new characters at specific position within the list called the "cursor index". The cursor index can be moved forward or backward.

The basic facilities of the ADT are defined as follows:

```
/**
 * A cursor is a pair (index, values), where values is list of integers
 * and index is an integer satisfying 0 <= index <= len(values).
 */
export interface MutableIntCursor {

  /** @returns index, where obj = (index, values) */
  index: () => number;

  /** @returns values, where obj = (index, values) */
  values: () => List<number>;

  /**
   * Inserts the given integer at the cursor index and moves the
   * cursor index forward by one.
   * @param m The integer to insert after the cursor index.
   * @modifies obj
   * @effects obj = (index + 1, concat(P, cons(m, S))),
   *     where (P, S) = split(index, values) and (index, values) = obj_0
   */
  insert: (m: number) => void;

  // ... more methods ...
}
```

Note that the specification of the `insert` method uses the "split" function from HW5. The definitions of this and other familiar list functions are provided on the final page of the exam.

As we saw in HW4, a list of integers can be used to represent text by storing character codes, which are integer values that identify specific characters. The following ADT implements the `MutableIntCursor` interface by using the abstract state (an index and a list of values) as its concrete state but by also recording the number of newline characters. That makes it easy for the class to quickly determine the number of lines in the text.

```
// The code of the newline character.
const newLine = "\n".charCodeAt(0);

class LineCountingCursor implements MutableIntCursor {
  // RI: 0 <= this.index <= len(this.values) and
  //     this.numNewlines = count(this.values, newline)
  // AF: obj = (this.index, this.values)
  index: number;
  values: List<number>;
  numNewlines: number;

  constructor(index: number, values: List<number>) {
    this.index = index;
    this.values = values;
    this.numNewlines = count(this.values, newline);
  }

  // ... methods implemented later ...
}
```

The representation invariant requires that `this.index` refers to a valid position in the list `this.values` and that `this.numNewlines` stores the number of newlines in `this.values`, which we can define formally using recursion:

$$
\begin{array}{llll}
\textbf{func}\ \mathsf{count}(\mathsf{nil}, c) & := 0 & & \text{for any } c : \mathbb{Z} \\
\quad \mathsf{count}(\mathsf{cons}(a, R), c) & := \mathsf{count}(R, c) + 1 & \text{if } a = c & \text{for any } a, c : \mathbb{Z} \text{ and } R : \mathsf{List} \\
\quad \mathsf{count}(\mathsf{cons}(a, R), c) & := \mathsf{count}(R, c) & \text{if } a \neq c & \text{for any } a, c : \mathbb{Z} \text{ and } R : \mathsf{List}
\end{array}
$$

Finally, the class will have the following factory function:

```
/**
 * Returns a cursor with the given abstract state.
 * @returns the cursor (index, values)
 */
export const makeLineCountingCursor =
    (index: number, values: List<number>): MutableIntCursor => {
  return new LineCountingCursor(index, values);
};
```

1. **Line-Craft**

Consider the following code, which claims to implement `insert` in `LineCountingCursor`:

```
insert = (m: number): void => {
  {{ Pre: this.numNewlines_0 = count(this.values_0, newline) }}
   const [P, S] = split(this.index, this.values);
   this.values = concat(P, cons(m, S));
  {{ Pre and ____________________________________________ }}
   this.index = this.index + 1;
   {{ Pre and ____________________________________________
     ______________________________________________________ }}
   if (m === newline) {
     {{ Pre and __________________________________________
       ____________________________________________________ }}
     this.numNewlines = this.numNewlines + 1;
     {{ Pre and __________________________________________
       ____________________________________________________
       ____________________________________________________ }}
   }
   {{ Post: this.index = this.index_0 + 1 and this.values = concat(P, cons(m, S))
              and this.numNewlines = count(this.values, newline)
          where (P, S) = split(this.index_0, this.values_0) }}
};
```

(a) Use **forward** reasoning to fill in the blank assertions above, which go into the "then" branch of the `if` statement. It is okay to use **subscripts** to refer to the original values of `this.index` and `this.values` (as is done in the postcondition).

Remember that constant values do not need to be tracked line-by-line, but those facts are available to us when we prove that the postcondition holds.

(Continued on the next page...)

(b) Explain, in English, why the fact listed in **Pre** will be true when the function is called.

(c) Explain, in English, why the facts listed in **Post** need to be true when the function completes in order for `insert` to be correct.

(Continued on the next page. . .)

(d) Prove by calculation the third fact of **Post** follows from the facts you wrote in the last blank assertion and the known values of the constants. Note that the values on the right-hand side of the constant declaration refer to the *original* values in those fields, not necessarily their current values!

(To be fully correct, we would also need to prove the first fact and do a similar analysis for the "else" branch, but we will skip those parts for this practice problem.)

You should also use[1] the following facts in your calculation:

- Lemma 1: $\text{concat}(P, S) = \text{this.values}_0$, where $(P, S) = \text{split}(\text{this.index}_0, \text{this.values}_0)$
- Lemma 5: $\text{count}(\text{concat}(L, R), c) = \text{count}(L, c) + \text{count}(R, c)$ for any c, L, R

[1] Extra practice problem: prove this claim by induction on L

2. **Hope For the Best, Prepare For the First**

   Fill in the missing parts of the following method so that it is correct with the **given invariant**.

   The **loop idea** is to skip past elements in `this.values` until we reach one that equals the given number or we hit the end. The first line of the invariant says that `this.values` is split up between `skipped` and `rest`, with `skipped` being the front part in reverse order. The second line of the invariant says that no element of `skipped` is equal to the number `m`.

   Do not write any other loops or call any other methods. The only list functions that should be needed are `cons` and `len`.

```
// Move the index to the first occurrence of m in values.
moveToFirst = (m: number): void => {
  let skipped: List<number> = ________________________;
  let rest: List<number> = ___________________________;

  // Inv: this.values = concat(rev(skipped), rest) and
  //      contains(m, skipped) = false
  while (_____________________________________________) {










  }

  if (rest === nil) {
    throw new Error(‘did not find ${m}‘);
  } else {
    this.index = ____________________________________;
  }
};
```

3. **Speech-to-Next**

Fill in the body of the `removeNextLine` method so that it removes all the text on the next line, i.e., the text between the first and second newline characters **after** the cursor index, along with the second newline character, but leaving the cursor index in place. If there are no newline characters after the cursor, then this should do nothing. If there is only one newline character after the cursor, this should remove all the text after that newline.

This is a method of `LineCountingCursor`, so you can access the fields `this.index` and `this.values`. You can call any of the Familiar List Functions on the final page and assume that each has been translated to TypeScript.

Hint: the split-at function from HW5 may be useful here. Assume the TypeScript translation of it is called `splitAt`.

```
// Removes the line of text after the one containing the cursor index
removeNextLine = (): void => {




};
```

Familiar List Functions

The function len(L) returns the length of the list L:

$$
\begin{aligned}
\textbf{func}\ & \mathsf{len}(\mathsf{nil}) && := 0 \\
& \mathsf{len}(\mathsf{cons}(x, L)) && := \mathsf{len}(L) + 1 && \text{for any } x : \mathbb{Z} \text{ and } L : \mathsf{List}
\end{aligned}
$$

The function concat(S, R) returns a list containing the values of S followed by those of R:

$$
\begin{aligned}
\textbf{func}\ & \mathsf{concat}(\mathsf{nil}, R) && := R && \text{for any } R : \mathsf{List} \\
& \mathsf{concat}(\mathsf{cons}(x, L), R) && := \mathsf{cons}(x, \mathsf{concat}(L, R)) && \text{for any } x : \mathbb{Z} \text{ and } L, R : \mathsf{List}
\end{aligned}
$$

The function rev(L) returns a list containing the values of L in reverse order:

$$
\begin{aligned}
\textbf{func}\ & \mathsf{rev}(\mathsf{nil}) && := \mathsf{nil} \\
& \mathsf{rev}(\mathsf{cons}(x, L)) && := \mathsf{concat}(\mathsf{rev}(L), \mathsf{cons}(x, \mathsf{nil})) && \text{for any } x : \mathbb{Z} \text{ and } L : \mathsf{List}
\end{aligned}
$$

The function contains(a, L) determines whether a is present in list L:

$$
\begin{aligned}
\textbf{func}\ & \mathsf{contains}(a, \mathsf{nil}) && := \ \mathsf{false} && \text{for any } a : \mathbb{Z} \\
& \mathsf{contains}(a, \mathsf{cons}(b, L)) && := \ (a = b) \text{ or } \mathsf{contains}(a, L) && \text{for any } a, b : \mathbb{Z} \text{ and } L : \mathsf{List}
\end{aligned}
$$

The function split(m, L) attempts to return a pair of lists (P, S), with P containing the first m characters from L and S containing the remaining characters from L.

$$
\begin{aligned}
\textbf{func}\ & \mathsf{split}(0, L) && := \ (\mathsf{nil}, L) && \text{for any } L : \mathsf{List} \\
& \mathsf{split}(m+1, \mathsf{nil}) && := \ \mathsf{undefined} && \text{for any } m : \mathbb{N} \\
& \mathsf{split}(m+1, \mathsf{cons}(a, L)) && := \ (\mathsf{cons}(a, P), S) && \text{for any } m : \mathbb{N},\ a : \mathbb{Z}, \text{ and } L : \mathsf{List} \\
& && \quad \text{where } (P, S) := \mathsf{split}(m, L)
\end{aligned}
$$

If $m \leq \mathsf{len}(L)$, split returns (P, S) with $\mathsf{len}(P) = m$ and $\mathsf{concat}(P, S) = L$.

The function split-at(L, c) always splits the given list L into a pair of lists (P, S), so that we have concat$(P, S) = L$. However, in this case, we are promised that P contains no c's, and S either starts with c or is nil. The function is defined formally as follows:

$$
\begin{aligned}
\textbf{func}\ & \mathsf{split\text{-}at}(\mathsf{nil}, c) && := \ (\mathsf{nil}, \mathsf{nil}) && && \text{for any } c : \mathbb{Z} \\
& \mathsf{split\text{-}at}(\mathsf{cons}(a, R), c) && := \ (\mathsf{nil}, \mathsf{cons}(a, R)) && \text{if } a = c && \text{for any } a, c : \mathbb{Z} \text{ and } R : \mathsf{List} \\
& \mathsf{split\text{-}at}(\mathsf{cons}(a, R), c) && := \ (\mathsf{cons}(a, P), S) && \text{if } a \neq c && \text{for any } a, c : \mathbb{Z} \text{ and } R : \mathsf{List} \\
& && \quad \text{where } (P, S) = \mathsf{split\text{-}at}(R, c)
\end{aligned}
$$