CSE 331: Software Design & Implementation

Section 7

1. The Right Search But the Wrong Pew

In this problem, we will check the correctness of code for binary search.

The goal of the function is to find the index "i" at which a value "x" could be inserted into the array A, which is currently sorted, and preserve its sorted order. Formally, our goal is to find the index i such that

$$A[k] \leq x \text{ for all } k = 0 \,..\, i-1 \text{ and } x < A[k] \text{ for all } k = i \,..\, n-1$$

where "n" is shorthand for A.length. This assertion says that every value in $A[0], \ldots, A[i-1]$ is less than or equal to x and every value in $A[i], \ldots, A[n-1]$ is greater than x, which means inserting x at this location would preserve sorted order.

(Note that we have a "$\leq$" in the left part and a "$<$" in the right part. In the example from lecture, those were swapped. The result of this change is that, if multiple entries of A contain the value x, this code will find the *last* index where x appears (at $A[i-1]$), whereas the version from lecture found the *first* index.)

Consider the following code, which claims to find the index:

$\{\{\ A[0] \leq A[1] \leq \cdots \leq A[n-1]\ \}\}$

```
let i: number = 0;
let j: number = A.length;
```

$\{\{\ P_1 :$ _____ $\}\}$

$\{\{$ Inv: $A[k] \leq x$ for all $k = 0 \,..\, i-1$ and $x < A[k]$ for all $k = j \,..\, n-1\ \}\}$

```
while (i !== j) {
    const m = Math.floor((i + j) / 2);
    if (A[m] <= x) {
```

$\{\{$ _____ $\}\}$

```
        i = m + 1;
```

$\{\{\ P_2 :$ _____ $\}\}$

```
    } else {
```

$\{\{$ _____ $\}\}$

```
        j = m;
```

$\{\{\ P_3 :$ _____ $\}\}$

```
    }
}
```

$\{\{\ P_4 :$ _____ $\}\}$

$\{\{$ Post : $A[k] \leq x$ for all $k = 0 \,..\, i-1$ and $x < A[k]$ for all $k = i \,..\, n-1\ \}\}$

The invariant, in this case, is a weakening of this post condition. It requires that everything in $A[0], \ldots, A[i-1]$ is less than or equal to x, but now, instead of requiring everything from $A[i], \ldots, A[n-1]$ is greater than x, it only requires that elements in $A[j], \ldots, A[n-1]$ are greater than x. All the elements in $A[i], \ldots, A[j-1]$ are *unconstrained* by this invariant: they could be either larger or smaller than x.

(a) Use **forward** reasoning to fill in P_1, before the loop; P_2 and P_3 (the postconditions of the two branches) inside the loop; and P_4, after we exit the loop. **You are permitted to use subscripts in this problem.**

Note that, since "m" is constant, you do not need to include its value in the assertions: it's value does not change line-to-line. The array A will also not be mutated by the code, so you do not have to copy the precondition in each assertion, we'll just remember that A is in sorted order. Also note that, if you want to repeat the exact loop invariant later on, you can just write "Inv" rather than copying the whole thing again. (When the invariant changes, though, you'll need to write it again.)

(b) Prove that P_1 implies Inv, P_2 implies Inv, P_3 implies Inv, and P_4 implies the postcondition.

(c) (If you have time:) How do we know that this loop actually terminates? Show that $j - i$ decreases every iteration.

2. Great Finds Think Alike

In this problem, we will implement the following function:

```
// Returns an index i such that words[i] === w or -1 if no such index exists
const findWord = (words: string[], w: string): number => {..};
```

We will implement this with a loop, but there are many different ways to do so. Each of these uses a slightly different invariant or increments the loop variable in a slightly different place. As you will show, any of these options can be made to work, so in practice, you are free to use whichever approach seems most natural to you.

(a) Fill in the missing parts so that the code is correct with the invariant provided.

```
let i: number = ____________; // fill this in
{{ Inv: there is no index 0 ≤ j ≤ i such that words[j] = w }}
while (____________________________) {        // fill this in

                                              // fill this in


   i = i + 1;
}
{{ there is no index 0 ≤ j ≤ words.length − 1 such that words[j] = w }}
return -1;
```

(b) Fill in the missing parts so that the code is correct with the invariant provided.

This invariant is the same as before, but we have moved the line where `i` is incremented.

```
let i: number = ____________; // fill this in
{{ Inv: there is no index 0 ≤ j ≤ i such that words[j] = w }}
while (____________________________) {        // fill this in
   i = i + 1;

                                              // fill this in


}
{{ there is no index 0 ≤ j ≤ words.length − 1 such that words[j] = w }}
return -1;
```

(c) Fill in the missing parts so that the code is correct with the invariant provided.

This version leaves the line where `i` is incremented as before, but now the invariant is changed.

```
let i: number = ____________; // fill this in
{{ Inv: there is no index 0 ≤ j ≤ i − 1 such that words[j] = w }}
while (____________________________) {        // fill this in
   i = i + 1;

                                               // fill this in

}
{{ there is no index 0 ≤ j ≤ words.length − 1 such that words[j] = w }}
return -1;
```

(d) Fill in the missing parts so that the code is correct with the invariant provided.

This invariant is the same as in (c), but we have moved the line where `i` is incremented back to the bottom of the loop body.

```
let i: number = ____________; // fill this in
{{ Inv: there is no index 0 ≤ j ≤ i − 1 such that words[j] = w }}
while (____________________________) {        // fill this in

                                               // fill this in

   i = i + 1;
}
{{ there is no index 0 ≤ j ≤ words.length − 1 such that words[j] = w }}
return -1;
```

3. Going Spaces

The following function takes an array of characters and returns a new array that is the same except that all space characters are removed (remember that there are not `chars` in TypeScript, so we represent characters with `strings` of length 1.):

$$
\begin{array}{lll}
\textbf{func}\ \text{del-spaces}([]) & := & [] \\
\text{del-spaces}(L \mathbin{+\!\!+} [\text{“ ”}]) & := & \text{del-spaces}(L) \\
\text{del-spaces}(L \mathbin{+\!\!+} [c]) & := & \text{del-spaces}(L) \mathbin{+\!\!+} [c] \quad \text{if } c \neq \text{“”}
\end{array}
$$

The most reasonable way to implement this function in TypeScript would be to take a string as input and return a new string. Strings are effectively immutable arrays in JavaScript, so we can treat them like arrays as long as we are only reading from them. Inside the function, we would need to build up the result in an array so that we can `push` elements on to it. We can convert the result to a string at the end by calling `.join("")` on the array.

However, in this problem, we will instead consider a version that takes an array of characters as input and mutates that array in-place.

(a) Write a specification for a function with the following signature that mutates the array passed in.

```
const delSpaces = (str: string[]): void => {..};
```

Next, we will write a loop that overwrites str with del-spaces(str) in-place.

More specifically, we will have a local variable "`m`" and have our loop establish the post-condition that str$[0 \,..\, m-1]$ = del-spaces(str$_0$), i.e., the first m elements of str now store del-spaces(str$_0$).

(b) How can we weaken that post-condition into something that is easy to make true at the top of the loop? What exit condition will tell us that the loop holds at the end.

(c) Implement the loop with the invariant above.

Be sure to document your loop invariant. Your code must be correct with that invariant.

(d) (If you have time:) The solution implicitly assumes that $m \leq$ str.length. Otherwise, we would not be able to write into the array at those locations. How would we prove that this inequality holds?

To complete the function, we need to remove the extra elements at the end. The following loop does that:

{{ Inv: str$[0 \,..\, m-1] =$ del-spaces(str$_0$) and $m \leq$ str.length }}

```
while (m !== str.length) {
    str.pop();
}
```

{{ str = del-spaces(str$_0$) }}

We only enter the loop if $m <$ str.length, and all pop does is reduce str.length by 1. (Importantly, the first part of the invariant doesn't say anything about what is in "str$[m \,..\, \text{str.length} - 1]$", so it is okay to remove these.)

When we exit, we have $m =$ str.length, so the invariant tells us that

$$\text{del-spaces}(\text{str}_0) = \text{str}[0 \,..\, m-1] = \text{str}[0 \,..\, \text{str.length} - 1] = \text{str}$$

The final assertion is the complete postcondition for the function. (Compare to your spec.)

4. Route For Joy

In this problem, we will write some code that runs in the server rather than in the browser (the "client"). Inputs will be passed in the query parameters as usual, but the response will be a JavaScript object serialized into text format ("JSON") and sent back to the browser. This is the most convenient format for a JavaScript client to parse back into an object.

To get started, check out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-23au-materials/sec-chatbot.git
```

Installing the modules with `npm install`, then start the server with the command `npm run start` and point your browser at `http://localhost:8080`. You should see a question on the screen. Unfortunately, it always asks the same question "What is your favorite color?" We would like it to return some more interesting questions.

Some more interesting questions are provided in `trivia.ts`. The `TRIVIA` list contains a number of records, each of which has a question and the answer. (Feel free to add more.)

(a) Change the `NewQuestion` function in `routes.ts` to return a randomly selected question from `TRIVIA`.

You can pick a random index from the list by calling `Math.random()`, which returns a number x with $0 \leq x < 1$, multiply it by the length of `TRIVIA` and then round down to an integer. You can then retrieve that integer index of `TRIVIA` using the "`at`" function found in `list.ts`.

Once you have done that, restart the server and see that a question `TRIVIA` is now shown.

You can also open up the "Network" tab in the developer tools and then, once you refresh the page, you will see every request sent from the browser to the server. One of these should be to the URL "`/new`", which will cause the server to call your `NewQuestion` function (see `index.ts`). If you click on this request, you should be able to see every detail of what was sent to and from the server.

(b) The app currently tells you that every answer is wrong. To see why, have a look at the `CheckAnswer` function in `routes.ts`. It currently returns `{correct: false}`, ignoring the answer the user gave us!

Change this code to check their answer correctly. You can use the "at" function to find the record of the question with the index that was passed in. You should ignore the casing (upper vs lower) when checking the user's answer matches the correct one.

Restart the server and check that it now tells you whether the answer was correct.

(c) Have a look at the tests in `routes_test.ts`. These are checking that we are sending back correct-looking responses for different user requests.

Run the tests using the command `npm run test`. Confirm that they all now pass.

(d) Have a look at `index.ts`. This file sets up the HTTP server using an object called app in the code. The call to `app.get("/check", CheckAnswer)` tells the HTTP server to answer requests for the url `/check` by invoking the function `CheckAnswer`. This mapping (of `/check` to `CheckAnswer`) is sometimes called a "route", and we are configuring app to set up the appropriate routes.

Even though we do have tests for `CheckAnswer` in `routes_test.ts`, notice that there is no `index_test.ts` file! What functionality that we implemented is not currently being tested? Give examples of bugs that could be in the code? Why is manual testing sufficient to catch such bugs? (Note that we do not need to test the HTTP server. That code base should have its own tests.)