CSE 331: Software Design & Implementation

Homework 1 (due Wednesday, October 4th at 11:00 PM)

In this assignment, you will implement a simple web application and gain experience with TypeScript and HTML.

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

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

Install the required libraries, run the command `npm install --no-audit` in the `hw-fib` directory.

The initial application does nothing at all. (You will add all of its functionality below.) However, the provided source code includes the function `fib` defined in `fib.ts` with the following shape:

```
export const fib = (n: number): number => { .. };
```

This function takes a non-negative integer n as input and returns the n-th Fibonacci number, denoted f_n.

In case you have not seen Fibonacci numbers before, they are defined as follows. The first two Fibonacci numbers are defined to be $f_0 = 0$ and $f_1 = 1$. For $n \geq 2$, the n-th Fibonacci number is defined *recursively* as $f_n = f_{n-2} + f_{n-1}$. (Note that this formula only makes for $n \geq 2$.) The `fib` function provided calculates the n-th Fibonacci number using exactly these formulas.Have a look at the code and make sure it all makes sense.

The provided code also includes a check to make sure that the user passed in a non-negative number. This is not strictly necessary, but it is a good idea to include it to help catch any bugs in other parts of the code. Writing such checks is called “defensive programming”.

The starter code also includes a file `fib_test.ts` with tests for the the `fib` function. We will learn more about testing in future assignments, but for now, note that each call to `assert.deepStrictEquals` passes in two numbers: the actual return value of `fib(n)`, for some `n`, and the expected answer. The function `assert.deepStrictEquals` checks that the two numbers are the same. If not, it prints an error message.

Confirm that the `fib` function provided passes the tests provided by running the command `npm run test`. You will see that the tests for some other functions (e.g., `fastFib`) are failing because those functions are not yet implemented. You will implement them later in the assignment.

1. Nail Our Colors to the Fast (20 points)

The provided `fib` function is a direct translation of the mathematical definition of Fibonacci numbers into TypeScript, making it easy to understand. Unfortunately, it is unacceptably slow, running in time $\Theta(2^n)$.

In this problem, you will implement a faster version, `fastFib`, with the following shape:

```
export const fastFib = (n: number): FibPair => { ... };
```

This version also takes n as an input, but instead of returning just f_n, it returns something called `FibPair`. The latter is a record type defined as follows:

```
export type FibPair = {curFib: number, prevFib: number};
```

As you can see, each `FibPair` includes not only that particular Fibonacci number, in the `curFib` field, but also the previous Fibonacci number, in the `prevFib` field. For example, the 3nd Fibonacci number would be represented by the record `{curFib: 2, prevFib: 1}` since $f_3 = 2$ and $f_2 = 1$.

Note that `fastFib` requires an input n that is at least 1. This is because, for $n = 0$, there is no such thing as the previous Fibonacci number, so there isn't any sensible record to return.

(a) Implement the body of `fastFib` using recursion. Specifically, a call to `fastFib(n)`, if `n` is not a base case, should call `fastFib(n-1)`.

Use only `const` declarations. Do not mutate anything!

(b) Verify that the tests for `fastFib` now pass when you run `npm run test`.

2. Virtue Is Its Own Record (10 points)

Now that we have `fastFib`, there is no reason to use the old, slow version.

(a) Change the implementation of `fib` to work by calling `fastFib` instead.

Note that you cannot simply write "`return fastFib(n)`" for two reasons. First, `fastFib` does not accept all the inputs that `fib` does. Second, `fastFib` does not return a `number`, like `fib` is supposed to. (It returns a `FibPair`, which is a record, not a number.) Your implementation will need to address both of these issues.

(b) Verify that the tests for `fib` still pass you run `npm run test`.

3. The Odd Tuple[1] (15 points)

The function `fastFib`, above, used a record to store a given Fibonacci number paired with the previous one. In general, records and tuples provide equivalent functionality. This means that we could have used tuples to define our function instead. In this problem, we will do that.

The starter code includes the function `fastFib2` that is just like `fastFib` except that it returns the type `FibPair2` instead of `FibPair`. The new type is defined as follows:

```
export type FibPair2 = [number, number];
```

This type declaration says that a `FibPair2` is two numbers, but it doesn't say which is the current Fibonacci number and which is the previous one. The comments in the code indicate that the previous number goes first. For example, the 3nd Fibonacci number would be represented by the pair `[1, 2]`.

This is a nice example of how the type system, while useful, cannot find all bugs for us. In this case, it would correctly identify the error if we tried to return just 2 instead of `[1, 2]`, but it would not spot the error if we returned the two numbers in the wrong order, i.e., as `[2, 1]` rather that `[1, 2]`. It is important to understand which errors the type system does and does not catch. We need to be especially careful about the latter category (only).

(a) Implement the body of `fastFib2`. Like `fastFib` it should use recursion, contain only `const` declarations, and not mutate anything.

(b) Verify that the tests for `fastFib2` now pass when you run `npm run test`.

[1]For the record, "tuple" is actually pronounced "too-ple", but that was too hard to pun.

4. Rally the Loops (20 points)

The following Java function returns the smallest Fibonacci number that is greater than or equal to m:

```
public static int nextFib(int m) {
  int prevFib = 0;
  int curFib = 1;
  while (curFib < m) {
    int temp = prevFib;  // change (prevFib, curFib)
    prevFib = curFib;    //   from (fib(n-1), fib(n)) to (fib(n), fib(n+1))
    curFib = curFib + temp;
  }
  return curFib;
}
```

The loop works its way up through the Fibonacci numbers, storing each one in turn in the variable `curFib`, until it gets to the first one that is greater than or equal to `m`. In order to calculate the next Fibonacci number, it needs to also keep track of the Fibonacci number before `curFib`, which is stored in the variable `prevFib`.

(The property of this code that, at the top of the loop, `curFib` $= f_n$ and `prevFib` $= f_{n-1}$ for some n is called a "loop invariant". We will have a lot more to say about those later in the course.)

In this problem, you will write a *recursive* version of the function above in TypeScript. Unlike the loop above, our recursive function will not need to mutate any variables.

The starter code includes a function called `nextFib` that handles the case $m = 0$ by returning 0. For $m \geq 1$, it calls a function `nextFibHelper`, which you will implement recursively.

(a) Implement `nextFibHelper` so that it simulates the loop above using recursion.

In more detail, the function should call itself recursively on each subsequent Fibonacci number until it gets to the first Fibonacci number that is greater than or equal to `m`, which it then returns.

Instead of storing the state in local variables, the state of the loop is now recorded in the function arguments: `prevFib`, `curFib`, and `m`. A call to `nextFib(m)` will invoke `nextFibHelper(0, 1, m)`. Those arguments describe the state of the local variables the *first time* we get to the top of the loop. Your recursive calls should be made with arguments that go through increasingly larger Fibonacci numbers until you get to the first one that is greater than or equal to `m`, at which point the recursion should stop.

(b) Verify that the tests for `nextFib` now pass when you run `npm run test`.

(c) We can think of a call to `nextFibHelper(prevFib, curFib, m)` as asking, what would the loop above ultimately return if it entered the top of the loop with the three variables having the given values?

Explain in your own words why your recursive implementation answers this question.

5. No Rest For the Query (20 points)

So far, we have written functions that calculate Fibonacci numbers and related quantities, but users have no way of using them. In this problem, we will make `nextFib` available from a user interface.

The provided starter code in `index.tsx` creates the `Root` in which we will render the UI, but the provided UI simply prints out one Fibonacci number. To make this useful, we need to get input from the user. For now, we will have the user pass their input in using query parameters.

(a) Change the code to look for query parameters called "`firstName`", which can be any non-empty-string, and "age", which must be a non-negative integer. Have the code render an error message if either parameter is missing or invalid. (See the section worksheet for how to do this.[2])

(b) Change the code so that, if both parameters are provided and valid, it renders a message telling the user either that their age is a Fibonacci number (if it is one) or the number of years until their age will be a Fibonacci number (if it is not one).

For example, if the user's first name is Jesse and their age is 21, it could render a message like this:

```
Hi, Jesse! Your age (21) is a Fibonacci number!
```

whereas if their age was 19, it could render a message like this:

```
Hi, Jesse! Your age (19) will be a Fibonacci number in 2 years.
```

(c) Start the server by running the command the command `npm run start`. Then, point your browser at `http://localhost:8080/?firstName=Jesse&age=21` to check that it works as intended. Try changing the age to a non-Fibonacci number (like 19) as well.

Also be sure to try changing the first name! You could have accidentally written "Jesse" directly in the code. In that case, it would look correct if "Jesse" is the only name you tested.

6. Theres a Form Brewin' (15 points)

While query parameters work, they are not something that a typical Internet user would know how to use. In this problem, we will give the user a more natural way to enter this information.

(a) Change the code in `index.tsx` so that, if neither the first name nor the age parameters are provided, rather than rendering an error message, it renders the following HTML:

```
<form action="/">
  <p>Hi there! Please enter the following information:</p>
  <p>Your first name: <input type="text" name="firstName"></input></p>
  <p>Your age: <input type="number" name="age" min="0"></input></p>
  <input type="submit" value="Submit"></input>
</form>
```

The `<input>` tags create UI components that allow the user to enter information. The first one with `type="text"` creats a standard text box that allows the user to enter any string. The second one with `type="number"` and `min="0"` allows the user to enter non-negative numbers.

The final `<input>` tag with `type="submit"` creates a button labelled "Submit" that, when clicked, will redirect the page to a URL put together by adding query parameters to the URL in the `action=".."`

[2]Technically, the solution described there, using `parseInt`, allows non-negative, *fractional* values by rounding down to the nearest integer. You could instead use `parseFloat` and then verify the result is an integer in the manner described in lecture. However, we will not require that for this assignment.

attribute of the `<form>..</form>` tag containing the submit button. Specifically, it adds one query parameter for each of the (non-submit) `<input>` tags inside the `<form>..</form>`, with the parameter name coming from the `name=".."` attribute of the `<input>` tag and the parameter value coming from the value typed by the user into that input box.

In this case, if the user types "Jesse" into the first input box and "21" into the second input box, then clicking the submit button will redirect the page to "`/?firstName=Jesse&age=21`".

(Note that the `type="number"` and `min="0"` attributes on the age `<input>` tag do not remove the need to check that the query parameters are valid when present. A user can still type in any query parameters they want into the URL bar of the browser.)

Run the application and verify that navigating to `http://localhost:8080/` shows the form. Then, verify that you can fill out the form, click "Submit", and see the expected message from Problem 5. Pressing the back button should take you back to the form.

(b) Change the HTML shown when the user provides first name and age query parameters so that, in addition to the message described in Problem 5, it also includes an `<a>` tag, labelled "Start Over", that navigates back to the form.

Congratulations on completing your first web app of CSE 331!

You can complete the following extra credit problem if you want. Once you are done, follow the instructions on page 7 to submit your solution in GradeScope.

7. Extra Credit: Like Watching Class Grow (0 points)

The UI we created in the previous problem works, but it is pretty bare bones. For extra credit, you can improve the styling so that it looks more fanciful.

To get credit, you must add your styling in a file, `index.css`, rather than directly in the code. You will need to create that file yourself. Once you have done so, you can define classes in the file and then assign classes to tags using a `className=".."` attribute in the code. (Make sure you "`import './index.css'`" in `index.tsx` so that the CSS is included in the page produced by React.)

To get credit, you must change at least the following components of styling: background colors, foreground colors, fonts, borders, and margin or padding.

Make sure the final result looks reasonable. (It must be readable as well as fanciful.)

How to Turn In Assignments

All work will be turned in via Gradescope. For each assignment, you will turn in your written work and code separately. In this assignment, the written work includes only your answer to problem 4(c), but later assignments will include a larger proportion of written work.

You will turn in your written work to the "HW# Written" assignment on Gradescope. Make sure that any handwritten work is legible (and dark enough) for us to read. If the graders cannot read your solution, they cannot give you points. Also, when you turn in your written HW to Gradescope, please match each HW problem to the page with your work. If you do not do this, you may receive a point deduction.

You will turn in your code to the "HW# Code" assignment on Gradescope. You only need to submit the *final version* of each of the files you worked on in the assignment.

In the HW instructions, for each coding problem, we will identify which files that you will need to upload to Gradescope. It is crucial that you only upload the files that we instruct you to, or else you may not pass the autograder. For example, if HW1 Problem 30 says to upload file `example.ts` and Problem 60 says to upload `fun.ts`, you should only upload `example.ts` and `fun.ts` to Gradescope. You should not put `example.ts` and `fun.ts` into a folder, and then upload the folder to Gradescope, as this will cause you to fail the autograder.

Be sure to look for any error messages from the autograder when you submit, and be sure that you received all of the auto-graded points. If you did not, you can examine the error messages, identify the problem, and then submit a corrected solution. Make sure you **leave enough time** to fix any errors that identified by the autograder. Try to avoid submitting for the first time in the last few minutes before the assignment is due because that will not leave you enough time to fix any problems encountered.