CSE 331: Software Design & Implementation

Homework 7 (due Wednesday, May 17th at 11:00 PM)

The problems in this assignment contain a mix of written and coding parts. When you have completed them, submit your solutions in Gradescope. The written problems should be clearly labeled and submitted as a pdf to the "HW7 Written" assignment. The following completed files should be directly submitted for the coding portion to the "HW7 Coding" assignment: `number_set.ts` `sorted_set.ts`

For complete instructions for how to submit assignments in this course see the Homework Turn-in Guide.

The coding parts of this assignment, in problems 1, 3, 4, 5, build on each other. Certain parts of the given code will not be be used until future questions, so please follow the directions and `//TODO` comments carefully to work in order. We will not require you to turn in intermediate versions of your code, just the final version, but following every step is in your best interest.

1. Work Up a Set (18 points)

The following parts consist entirely of coding work. They should be submitted with "HW7 Coding".

Start by checking out the starter code using the command

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

Install the modules using `npm install --no-audit`, then try running the application with `npm run start`. The app allows the user to search for numbers with some combination of properties such as being "not even and prime and not fibonacci". It will then print out the numbers (between 1 and 100) satisfying that condition.

The application records information about sets of numbers in an array of booleans called a `NumberSet`. If S is such a set then $S[j]$ says whether j is a number in the set. Each `NumberSet` is an array of 101 booleans, which can indicate the presence of any number from 0 to 100.

(The implementation actually uses a variable `MAX = 100` instead of hardcoding 100 directly. This is used for flexibility in provided tests in a later problem. You can just think of it as only 0 to 100 and disregard it and the related function `setMaxForTesting`.)

This representation leaves a lot to be desired, most specifically that it cannot store information about numbers larger than 100. We will fix this by moving to a different representation, but first, we need to abstract the details of the representation — i.e., create an ADT — so that we have room to make that change later.

(a) Create a new interface `NumberSet` in `number_set.ts` that has just the operations, `removeAll`, `addAll`, and `getNumbers`. Be sure that the interface and its functions have proper documentation.

Look to the `removeAll`, `addAll`, and `getNumbers` functions already in the file for an idea of what these operations should be doing (ignore the `complement` function, for now). Note, however, that the new operations are part of an ADT, so they do not need to take the set that we are removing from/adding to/getting elements of as a parameter, as that set would be represented by the ADT itself.

(b) Create a class in `number_set.ts` called `BooleanNumberSet` that stores the set, as a `boolean[]`, in a field and implements the operations in the same way the current functions do.

Use the current functions as helper functions and modify them as you see fit –they will no longer be exported, so this is fine. Hint: in particular, you will need to modify the parameters of these helpers as `NumberSet` is no longer just a type consisting of a `boolean[]`.

We will not mix and match different types of `NumberSets` in the application, so you can safely cast the `NumberSets` passed as arguments into `BooleanNumberSets` using the notation "`as BooleanNumberSet`".

You can embed this cast into a function call/field access by doing: `(set as BooleanNumberSet).field` where `set` is the variable holding the `NumberSet`.

Be sure to properly document the abstraction function of the class. (Use English if there is no obvious notation for what you want to say.)

(c) Change the name of `makeNumberSet` to `makeBooleanNumberSet` since we will have different implementations of `NumberSet` in the future and have it return an instance of `BooleanNumberSet`.

Make sure that `NumberSet`, `makeBooleanNumberSet`, and `setMaxForTesting` are the only symbols exported in the file. Clients should not be able to directly access the class, the helper functions, or anything else.

(d) Change the tests in `number_set_test.ts` to use `makeBooleanNumberSet` to create an instance of `NumberSet` and use that instance to perform the operations throughout the tests.

Make sure that all the `number_set` tests still pass by running `npm run test`. (Others will still fail. We will fix those next.)

(e) Change the code in `routes.ts` to call the `.getNumbers()` method rather than the old function `getNumbers`, which is no longer exported. Do the same in `eval_test.ts`.

(f) Change the code in `eval.ts` to use the factory function instead of the old `makeNumberSet`. For now, ignore the `TODO`s about question 5.

Confirm that all the tests (besides "uniquify") now pass.

(g) Make sure that the app still works by running `npm run start` and trying it out.

2. From Loop to Nuts (24 points)

The following parts consist entirely of written work. They should be submitted with "HW7 Written".

In this problem, we will check the correctness of a loop that implements "without(L, R)" from quiz section. Unlike the recursive version, which returns a new array, this version works "in place", replacing L with the result.

Our precondition includes the facts that L and R are both sorted and contain distinct elements, i.e., that $L[0] < L[1] < \cdots < L[n-1]$ and $R[0] < R[1] < \cdots < R[m-1]$, where n and m are the lengths of L and R.

We will use the bottom-up pattern (on L) to calculate the result. The variables "`i`" and "`j`" will keep track of how much of L and R, respectively, that we have processed so far. The variable "`k`" will keep track of how much of L has been overwritten with the new result.

Formally, the invariant has three parts:

1. $L[0 \,..\, k-1] = \text{without}(L_0[0 \,..\, i-1], R)$, which formalizes what we said above
2. $L[k \,..\, n-1] = L_0[k \,..\, n-1]$, which says that the rest of L is unchanged
3. $R[j-1] < L[i]$, where those indexes exist[1]

Item 3 formalizes what we learned in quiz section about the recursive version: at any point, we only care about the part of R that is less than or equal to the last element of L. Here, the index "`j`" keeps track of the part of R that still matters. As we will see, we only need to compare $L[i]$ to $R[j]$ in order to figure out how to calculate without$(L[0 \,..\, i], R)$ from without$(L[0 \,..\, i-1], R)$.

[1]I.e., $j - 1 < 0$ or $i = n$, then the inequality should be ignored.

(Technically, we should also include the fact that $k \le i$, but it is straightforward to see that this holds, so we will skip checking it explicitly.)

Our postcondition will be that $L[0 \,..\, k-1] = \text{without}(L, R)$. Note that, if $k < n$, then L still contains some of the old elements of L_0 at the end. However, these are easy to remove with another loop. We will skip that part for this problem.

```
let i: number = 0;
let j: number = 0;
let k: number = 0;
{{ $P_1$ : ________________________________ }}
{{ Inv: $L[0 \,..\, k-1] = \text{without}(L_0[0 \,..\, i-1], R)$ and $L[k \,..\, n-1] = L_0[k \,..\, n-1]$
     and $R[j-1] < L[i]$ }}
while (i !== L.length) {
  if ((j === R.length) || (L[i] < R[j])) {
    {{ $P_2$ : ________________________________ }}
    {{ $Q_2$ : ________________________________
           ________________________________ }}
    L[k] = L[i];
    i = i + 1;
    k = k + 1;
  } else if (L[i] > R[j]) {
    {{ $P_3$ : ________________________________ }}
    {{ $Q_3$ : ________________________________
           ________________________________ }}
    j = j + 1;
  } else {
    {{ $P_4$ : ________________________________ }}
    {{ $Q_4$ : ________________________________
           ________________________________ }}
    i = i + 1;
    j = j + 1;
  }
}
{{ $P_5$ : ________________________________ }}
{{ $Q_5$ : $L[0 \,..\, k-1] = \text{without}(L_0, R)$ }}
```

(a) Use reasoning to fill in all blank assertions above. The 'P_i's should be filled in with forward reasoning and the 'Q_i's should be filled in with backward reasoning. Though you don't need to turn it in, you may find it useful to write intermediate assertions on scratch paper when they jump over multiple lines of code.

Write your assertions with mathematical definitions and notations rather than code notation. For example, instead of referring to L.length, we'll denote it as 'n' and instead of referring to R.length, we'll denote it as 'm'. If you want to repeat the exact loop invariant in other assertions, feel free to just write 'Inv.' However, if any part of the invariant changes you should rewrite it.

For all of these, remember that we learned in quiz section, if $L[i-1] < R[j]$, then we know that without$(L[0 \,..\, i-1], R) =$ without$(L[0 \,..\, i-1], R[0 \,..\, j-1])$, i.e., the rest of R does not matter.

(b) Prove that P_i implies Q_i for $i = 1, 2, 3, 4, 5$ (with Q_1 being "Inv").

If you are having a hard time proving a statement holds starting from the left side of the $=$ sign, try starting from the right side instead, or vice versa.

3. Loop Therapy (18 points)

The following parts consist of a mix of written and coding work: parts (f, g) are written and should be submitted with "HW7 Written", while parts (a,b,c,d,e) are coding and should be submitted with "HW7 Coding".

Our original implementation of `NumberSet` used an array to record which numbers were included. Among other problems, this representation can only represent subsets of the numbers between 1 and 100. In this problem, we will create a representation that stores the numbers in a sorted array and eliminates that problem.

The functions with and without are already provided in `sorted_set.ts`, where they are called `addAll` and `removeAll`. (We couldn't call it "with" since that is a reserved word in JavaScript.)

The other function we will need is one that removes duplicates from a sorted array (because with and without assumed no duplicates as well as being sorted). We can define a function that does this as follows:

$$
\begin{array}{llllll}
\textbf{func} & \text{uniquify}([\,]) & := & [\,] & & \\
& \text{uniquify}([a]) & := & [a] & & \text{for any } a : \mathbb{Z} \\
& \text{uniquify}(L +\!\!+ [a,b]) & := & \text{uniquify}(L +\!\!+ [a]) +\!\!+ [b] & \text{if } a < b & \text{for any } a, b : \mathbb{Z} \text{ and } L : \text{Array}_{\mathbb{Z}} \\
& \text{uniquify}(L +\!\!+ [a,b]) & := & \text{uniquify}(L +\!\!+ [a]) & \text{if } a = b & \text{for any } a, b : \mathbb{Z} \text{ and } L : \text{Array}_{\mathbb{Z}} \\
& \text{uniquify}(L +\!\!+ [a,b]) & := & \text{undefined} & \text{if } a > b & \text{for any } a, b : \mathbb{Z} \text{ and } L : \text{Array}_{\mathbb{Z}}
\end{array}
$$

The function returns undefined in this last case because the input array is not sorted as it should be.

It is possible to prove that contains$(L, x) =$ contains(uniquify$(L), x)$, for any sorted array L and integer x, i.e., that uniquify(L) contains the same set of numbers as in the original sorted array, and that it also contains no duplicates. (See the extra credit problem for details.)

(a) Implement the missing parts of `uniquify` in `sorted_set.ts` so that it implements the function above using a loop and operates in-place (without needing another array).

The invariant for the main loop is already provided. It references local variables "`i`", which keeps track of how much of the input array "`vals`" has been processed, and "`k`", which keeps track of where the result has been stored in `vals`. The invariant has three parts, like those of without from earlier:

1. $L[0 \,..\, k-1] =$ uniquify$(L_0[0 \,..\, i-1])$, which says $L[0 \,..\, k-1]$ stores the result so far
2. $L[k \,..\, n-1] = L_0[k \,..\, n-1]$, which says that the rest of L is unchanged
3. $L[i-1] = L[k-1]$, which says the last elements are the same.

Note that, in this case, the indices in 3 must exist, i.e., $i, k \geq 0$ must always hold. (Item 3 is actually implied by item 1, but it's easier to reason through the code if we write this out explicitly, I think.)

After you have filled in the loop body, write code after the loop to ensure the postcondition holds.

Think carefully through the code. Try to get it right on the first try!

Make sure that the tests pass by running `npm run test`.

(b) Create a class in `sorted_set.ts` called `SortedNumberSet` that implements the `NumberSet` interface by storing the numbers in a sorted array.

Be sure to properly document the class, which should have both an AF and RI.

Use the given functions in `sorted_set.ts` to implement the operations of the class and instantiate your fields in the constructor as needed. The factory function we create in the next step is the only way we will create instances of this class, so you have some flexibility in choosing what to pass to a `new SortedNumberSet`.

Note that JavaScript's Array class has a built-in `sort` method. You should definitely use that rather than trying to implement your own, but be sure to read the documentation on MDN carefully!

(c) Create a factory function, called `makeSortedNumberSet`, with the declaration below. Be sure to write a specification for the function.

```
export function makeSortedNumberSet(vals: List<number>): NumberSet
```

(d) Uncomment the tests for `makeSortedNumberSet` in `sorted_set_test.ts` (leave the tests for later questions commented out). Note that `removeAll`, `addAll`, and `uniquify` are already tested, so the only new tests we need for our class are for the factory function.

Confirm that all the tests still pass by running `npm run test`.

(e) Change the code in `eval.ts` to use `makeSortedNumberSet` instead of `makeBooleanNumberSet`.

Confirm that all the tests still pass by running `npm run test`.

Make sure that the app still works by running `npm run start` and trying it out.

(f) The `getNumbers` method currently returns the numbers in the set in a `List<number>`. Since the numbers are stored internally in a `number[]`, we have to copy the numbers from the array, which takes $\Theta(n)$ time.

We could do this in constant time if we changed the return type to `number[]` and then returned the sorted array field directly. Performance aside, in what other ways is this second design better or worse?

(g) Suppose that our representation stored the points in a `List<number>` instead of an array. In that case, we could also return the points in constant time just by returning the field directly.

How would your answer to the last problem change in that case?

4. See the Attached Complement (16 points)

The following parts consist entirely of coding work. They should be submitted with "HW7 Coding".

In this problem, we will be working with the `complement` of a set. The complement of a set S within the range a .. b, is the set of integers x, with $a \leq x \leq b$, that are *not* contained in S. For example, over the range 1 .. 5 the set {1, 4, 5} has the complement {2, 3}. Without the boundaries of a range, a finite set has a complement that is *infinite*. For example, the unbounded complement of {1, 4, 5} is {-∞, .., 0, 2, 3, 6, .., ∞}.

The code in `eval.ts` makes heavy use of the function `complement`. It is used to implement the "`not`" and "`and`" operations. Since it is frequently used, it is beneficial to incorporate this operation into our `NumberSet` ADT and to try to make it faster.

As we have shown with the separate `BooleanNumberSet` and `SortedNumberSet` implementations, there are multiple ways to represent an ADT. One reason you may choose to create a certain representation is because it makes a frequently-used operation fast. In this case, the `complement` operation can be performed extremely fast (constant time) with the `SortedNumberSet` implementation.

In this problem, we will add `complement` to our ADT and classes.

(a) Change the `NumberSet` interface to include a new operation called `complement` (and its documentation) that changes S into its complement (the set of numbers not in S).

Our ADT currently represents finite sets, but finding the complement means representing a set that is *infinite*. Since it is impossible to store every number in an infinite set, modify the `getNumbers` operation (and its documentation) to take arguments a and b and, if the set is an infinite set, it will return only integers x with $a \leq x \leq b$ that are in the set. If the set is a finite set, it will return the full contents of the set, but we will add the requirement that all values in the set must be between a .. b (the bounds must encompass the set).

Next, update `BooleanNumberSet`.

(b) Change `BooleanNumberSet` to implement `complement` by uncommenting the `complement` function already provided in `number_set.ts` and calling it.

We have already provided tests for this function in `number_set_test.ts`. Uncomment them now! (These tests are the reason we included `MAX` and `setMax` as they allow us to test the complement of a set with a smaller range.)

Change `getNumbers` to take additional arguments parameters. `BooleanNumberSet` is already defined on the bounds `0..MAX`, so we don't have the getting an infinite set issue. To ignore them, make the parameters unnamed by changing them to: `(_: number, __: number)`. Note that this deviates from our `getNumbers` specification in the `NumberSet` interface a bit; because `complement` is an exercise we care more about you completing for `SortedNumberSet`, we're breaking the spec here just to make things easier. If it doesn't seem like it does quite the right thing, that's because it doesn't :).

Change the tests in `number_set_test.ts` to pass in (1, 100) as the arguments for all calls to `getNumbers`.

Make sure that all `number_set` tests pass by running `npm run test`.

Next, we will change `SortedNumberSet` to represent either a finite set or the complement of a finite set. Again, the complement of a finite set is *infinite* which we cannot store all the elements of. So, after the `complement` operation is used on a set, our array will store the numbers that are *not* in the set, a finite number of values.

(c) Change the representation of `SortedNumberSet` to allow it to represent either a finite set or the (infinite) complement of a finite set.

Add a field, `comp: boolean`, that indicates if we're representing a complement. `comp` is false when we represent a finite set and true when we represent the complement of a finite set. In other words, `comp = true` indicates that the values we are storing in the `number[]` field are the values *not* in the infinite set we are representing. For example, $[1, 4, 5]$, `comp = false`, represents the set $\{$1, 4, 5$\}$, but $[1, 4, 5]$, `comp = true`, represents the complement: $\{$-∞, .., 0, 2, 3, 6, .., $\infty\}$.

Creating a `new SortedNumberSet` should still create a finite set as the default, so `comp` should be initialized as `false` in the constructor.

Update the abstraction function to explain what the assignment of `comp` tells us about the `number[]`.

(d) Implement the function `complement` by flipping the value of the `comp` field.

(e) Change `getNumbers` to take a and b as arguments. When `comp` is false, it'll behave as it used to. When `comp` is true, we need to calculate the elements between a and b that are in the complement (meaning all the elements between a and b that are *not* stored in our `number[]`). You may find it useful to create a new sorted array containing all the elements within the range, then remove all the elements that are *not* in the complement from it.

Document your loop, but note that convincing a reader that this loop works as intended may be better done with a simple English comment than a formal loop invariant.

Fix the tests in `sorted_set_test.ts` to pass bound arguments to all calls to `getNumbers` such that the $a \leq$ the smallest value and $b \geq$ the largest value in the array.

(f) Uncomment the given tests for `complement` in `sorted_set_test.ts`. Make sure they pass by running `npm run test`.

5. Boolean or Not, I'm Walking on Air (24 points)

The following parts consist entirely of coding work. They should be submitted with "HW7 Coding".

With `BooleanNumberSet`, adding `complement` did not change the representation invariant. So the other operations, `removeAll` and `addAll` are unaffected. (Their correctness only depends on their own specs, AF, and RI.) However, since we have changed the representation of `SortedNumberSet`, which now stores another field, those methods are actually now broken!

The tests are still passing because they only call `removeAll` and `addAll` when both of the sets are finite. We need to fix these functions so that they work properly when one or both of the sets involved are infinite.

(a) Change the implementation of `removeAll` to work properly to represent all combinations of finite and infinite sets. There are four possibilities in total, and the current code only works properly for one of them (both finite).

Here are some hints for how to handle the other cases[2]:

Take R and L to be finite sets. Remember if a set is finite, in our code, the field `comp` will be false. If we have found the complement of a set (all the integers *not* in it), then `comp` will be true.

- If we remove "all the integers not in R" from "all the integers not in L", we will get a finite set because the only numbers that can still be left at the end are those we can find within R, which must be finite since R itself is. All the numbers in L are already not in the set, so the ones that will be left are those that are in R but not in L.
- If we remove R from "all the integers not in L, we are left with all the integers not in L or R. The latter is "all the integers not in S", where S is a list containing all the numbers from either list.
- If we remove "all the integers not in R" from L, we are left with all the integers in both L and R. One way to calculate that the list of numbers in both lists is to start by calculating $S = \text{without}(L, R)$, the list of numbers in L but not R, and then calculate the list we want as $\text{without}(L, S)$. Dropping the numbers in only L and not R leaves the numbers present in both lists.

As usual, try to reason carefully through the code and get it right on the first try.

Also, note that, if you need to get a copy of an array `A`, you can do so with `A.slice(0)` in JavaScript.

(b) Uncomment the given tests for `removeAll - infinite` in `sorted_set_test.ts`. Make sure they pass by running `npm run test`. Notice that to test the behavior of this function on an infinite set, we first create the set, find its complement with `.complement()`, call the function `.removeAll()`, and then check that the set after these operations looks as we expect.

(c) Change the implementation of `addAll` to work properly for all combinations of finite and infinite sets.

Here are some hints for how to handle the other cases:

Take R and L to be finite sets.

- If we add "all the integers not in R" to "all the integers not in L", we get all the integers not in both L and R.
- If we add "all the integers not in R" to the integers in L, we get all the integers not in S, where S is a list containing the numbers in R but not L.

[2]Students who have taken 311 should find it simple to confirm all of these facts via set theory. 311 Set Definitions

- If we add the integers in R to "all the integers not in L", we get all the integers not in S, where S is a list containing the numbers in L but not R.

(d) Uncomment the given tests for `addAll - infinite` in `sorted_set_test.ts`. Make sure they pass by running `npm run test`. Again, notice the steps we took to create each test like for `removeAll`.

We can now change the application to use the new operation.

(e) Remove the `complement` function in `eval.ts`, and change the code to instead use the method.

Update `eval_test.ts` to pass arguments to `getNumbers`. Use the same values that are passed as the last two arguments to `evaluate` in the tests.

Update `routes.ts` to pass arguments to `getNumbers`.

Make sure that all the tests now pass by running `npm run test`.

(f) Remove code in `routes.ts` that checks that the range is 1–100, replacing it with the commented-out code that sanity checks arbitrary ranges.

You should now be able to change the range of numbers to search in the UI.

Make sure that the app works by running `npm run start` and trying it out.

Congratulations, you still have a fully functioning app, now built using efficient ADTs!

6. Extra Credit: Size In the Back of Your Head (0 points)

The following parts consist entirely of written work. They should be submitted with "HW7 Written".

Suppose that we added a `size` operation on `NumberSet`, defined as follows:

```
/**
 * Returns the size of the set of numbers.
 * @returns len(obj)
 */
size(): number
```

Note that, since we told clients to think of sets as lists of distinct numbers, we can define the size of the set to just be the length of that list.

Let's consider implementing this operation on `BooleanNumberSet`, which represents the set using an array `boolean[]`. Unfortunately, our notation makes it hard to talk about the index of an element in the array, so let's think of this mathematically as an array of pairs (i, b), where i is the index and b is true or false. If that is the case and we call the array of booleans "`included`", then we can define the abstraction function as `obj = values(this.included)`, where values is defined by:

$$\begin{aligned}
&\textbf{func } \mathsf{values}([]) &&:= [] \\
&\quad \mathsf{values}(L \mathbin{+\!\!+} [(i, \mathsf{false})]) &&:= \mathsf{values}(L) &&\text{for any } i : \mathbb{N} \text{ and } L : \mathsf{Array} \\
&\quad \mathsf{values}(L \mathbin{+\!\!+} [(i, \mathsf{true})]) &&:= \mathsf{values}(L) \mathbin{+\!\!+} [i] &&\text{for any } i : \mathbb{N} \text{ and } L : \mathsf{Array}
\end{aligned}$$

With those definitions in mind, suppose we implement `size` as follows:

```
size = () => { return size(this.included) };
```

where `size` is a direct translation of the following mathematical function:

$$\begin{aligned}
&\textbf{func } \mathsf{size}([]) &&:= 0 \\
&\quad \mathsf{size}(L \mathbin{+\!\!+} [(i, \mathsf{false})]) &&:= \mathsf{size}(L) &&\text{for any } i : \mathbb{N} \text{ and } L : \mathsf{Array} \\
&\quad \mathsf{size}(L \mathbin{+\!\!+} [(i, \mathsf{true})]) &&:= \mathsf{size}(L) + 1 &&\text{for any } i : \mathbb{N} \text{ and } L : \mathsf{Array}
\end{aligned}$$

(a) What fact needs to be true in order for the code above to be correct according to its specification (with this choice of abstraction function)?

(b) Prove by induction that $\mathsf{len}(\mathsf{values}(L)) = \mathsf{size}(L)$ holds for any list of pairs L in the form described above.

7. Extra Credit: Nothing to Contain About (0 points)

The following parts consist entirely of written work. They should be submitted with "HW7 Written".

In the third problem, we stated that "uniquify" returns an array that contains the same numbers but no duplicates. In this problem, we will prove the first of those claims.

In order to do so, we need this definition of "contains" for arrays (our previous definition was on lists):

$$\begin{aligned}
&\textbf{func } \mathsf{contains}([], y) &&:= \mathsf{false} &&\text{for any } y : \mathbb{Z} \\
&\quad \mathsf{contains}(L \mathbin{+\!\!+} [x], y) &&:= (x = y) \text{ or } \mathsf{contains}(L, y) &&\text{for any } x, y : \mathbb{Z} \text{ and } L : \mathsf{Array}_{\mathbb{Z}}
\end{aligned}$$

With that in hand, we can now prove the properties claimed.

Let y be any integer. Prove by induction that, for any **sorted** list L, we have

$$\mathsf{contains}(L, y) = \mathsf{contains}(\mathsf{uniquify}(L), y)$$