CpSc 418 **Homework 2** Due: October 6, 2017, 11:59pm
Early Bird: October 4, 2017, 11:59pm

147 points.

Please submit your solution using the handin program. Submit your solution as
cs418 hw2
Your submission should consist of three files:

- `hw2.erl`: Erlang source code for the coding parts your solution.
- `hw2.pdf` or `hw2.txt`: PDF or plain-text for the written response parts of your solution.
- `hw2_tests.erl`: EUnit tests for your solution.

A template for `hw2.erl` is available at
http://www.ugrad.cs.ubc.ca/~cs418/2017-1/hw/2/src/hw2.erl

Please submit code that compiles without errors or warnings. If your code does not compile, we will probably give you zero points on all of the programming problems. If we fix your code to make it compile, we will take off lots of points for that service. If your code generates compiler warnings, we will take off points for that as well, but not as many as for code that doesn't compile successfully. See the comments about errors, warnings, and guards at the end.

Overview

This homework turned out to be an adventure in the process of applying the reduce pattern to an "obvious" example. In the process, I stumbled across several places where the actual implementation teetered on the edge of becoming a mess. This is fairly common in parallel programming (and programming in general, for that matter). I decided to make the homework a sequence of exercises in taking an example of reduce, and making it work in practice.

One of the problems on Homework 1 was to write an efficient function for computing `fib(N)`, the N^{th} Fibonacci number. Let's say that we have a sequence of integers, x_1, x_2, ..., x_N. We will say that this sequence is a *Fibonacci sequence* iff for all $3 \leq i \leq N$,

$$x_i \quad = \quad x_{i-2} + x_{i-1}$$

Now, let's say we have some list, `L=[E1, E2, ...EN]`. We would like to find the longest sublist of `L` that is Fibonacci sequence as defined above. In particular, we would like to implement

```
longest_fib_seq(List) -> {StartPos, Length}
```

where `StartPos` is the index of the longest Fibonacci sequence in `List` and `Length` is the length of this sequence. If there are two or more sublists of `List` that are Fibonacci sequences of maximal length, `longest_fib_seq(List)` should return the *first* one, i.e., the one with the smallest value for `StartPos`. For example,

```
longest_fib_seq([42,17,59,1,3,4,7,11,2,13,15,28,1000])
```

should return `{4,5}`.

Sounds like an easy problem – what's the big deal? I encountered two issues. The first issue arose in the computation to be performed by each leaf. Unless the longest subsequence is the trivial one – a long

sequence of `0` values – then the magnitude of the elements of the Fibonacci sequence grows exponentially with the length of the sequence. This means the time for checking that a subsequence if a Fibonacci sequence grows quadratically with the length of the sequence due to the way big-integers are implemented. This isn't necessarily bad, we like problems that have much more computation than communication, but, it makes it hard to make test cases that aren't wildly imbalanced in the amount of work each task does. There are ways to handle such load balancing problems, but I didn't want that to be part of our first parallel programming assignment.

The second issue arises in the combine function. I tried several approaches, and I kept bumping into a fairly large number of corner cases. I want this homework to focus on reduce, and not on how to cover lots of corner cases using Erlang.

My solution to the issue of `fib(N)` being too large is to define a variation of `fib` that I'll call `fibm` as defined below

```
fibm(N, M) = fib(N) rem M.
```

The nice thing about this is we can perform the `rem M` operation at each step of computing the Fibonacci sequence. This guarantees that all intermediate results are in the interval $[0, \texttt{M} - 1]$ – for a reasonable choice of `M` (e.g. 100, or even 1,000,000,000), each arithmetic operation in the implementation of `fibm` calculation has "small" operands. We can compute `fibm(N,M)` in $O(N \log M)$ time. If `M` is an integer that can be represented with a single machine word, then we get $O(N)$ time. Hooray! For any questions involving the run-time of the functions you are supposed to implement, you can assume that `M` is small, e.g. $0 < \texttt{M} < 2^{32}$.

My solution to the problem of too many corner cases is to walk you through the design and provide code for parts that I deemed to be more tedious than enlightening. With that introduction, here are the questions.

This walk-through approach produced a long-document for the homework. Don't worry, you can think of it as a reading assignment with exercises included. So you won't lose track of where the questions are that you need to do, I've included arrows in the right margin pointing at the actual questions.

The Questions

1. `fibm` (10 points)
   We define `fibm(N, M)` as shown below.

   ```
   fibm(N, M) when is_integer(M), M > 0 -> fib(N) rem M.
   ```

   We note that we can perform the `rem M` operation at each step of the computation. For example,

   ```
   fibm(0, M) when is_integer(M), M > 0 -> 0;
   fibm(1, M) when is_integer(M), M > 0 -> 1;
   fibm(N, M) when is_integer(N), N > 0, is_integer(M), M > 0 ->
     fibm(N-2, M) + fibm(N-1, M).
   ```

   For example,

   ```
   fibm(0, 100) = 0;
   fibm(10, 100) = fib(10) rem 100 =  55 rem 100 = 55;
   fibm(11, 100) = fib(11) rem 100 =  89 rem 100 = 89;
   fibm(12, 100) = fib(12) rem 100 = 144 rem 100 = 44;
   fibm(42, 100) = fib(42) rem 100 = 267914296 rem 100 = 96.
   fibm(42, bananas) raises an error
   ```

Of course, this implementation has a run-time that is exponential in `N` just like `awful_fib` from Homework 1.

Write an efficient implementation of `fibm`. Provide test-cases in `hw2_tests.erl`. ← Q1

2. `is_fibm_seq` (10 points)
Let `List` be an Erlang list and let `M` be a positive integer. We say that the elements of `List` are a `M`-Fibonacci sequence iff

   - Every element of `List` is an integer.
   - For every `I` with `3 =< I` and `I =< length(List)`,
     `lists:nth(I, List) = (lists:nth(I-2, List) + lists:nth(I-1, List)) rem M`

   `is_fibm_seq(`*X*`)` should return either 'true' or 'false' (e.g. it should not raise an error) no matter what is given for *X*.

   Write an implementation of `is_fibm_seq`. The runtime of `is_fibm_seq(List, M)` should be $\mathcal{O}$(`length(List)`). ← Q2

3. `longest_fibm_seq`, sequential version (25 points)
Let `List` be a list of integers and let `M` be a positive integer. Find the longest subsequence of `List` that is a `M`-Fibonacci sequence as defined in Question 2.

   (a) (15 points) Write the code. ← Q3.a
   Write a function

   `longest_fibm_seq(List) -> {`*StartPos*`, `*Length*`}.`

   where *StartPos* is the position in the list of the first element of the longest `M`-Fibonacci sequence of `List`, and *Length* is the length of this subsequence. If there are two or more `M`-Fibonacci sequence of the same, maximum length, then *StartPos* should refer to the *first* one. If `List` has any elements that are not integers, then `longest_fibm_seq(List)` should raise an error. For example,

```
longest_fibm_seq([0,1,1], 1000) -> {1,3};
longest_fibm_seq([0,1,2,3,5,10,15,25,40,65,100,200], 1000) -> {5,6};
longest_fibm_seq([0,1,2,4,6,10,10,20,30,40,70,110], 1000) -> {3,4};
longest_fibm_seq([5,4,3,2,1,0], 1000) -> {1,2};
longest_fibm_seq([5], 1000) -> {1,1};
longest_fibm_seq([], 1000) -> {1,0};
longest_fibm_seq([42, bananas], 1000) -> % raises an error.
```

   (b) (5 points) Run-time analysis. ← Q3.b
   Let $N =$ `length(List)`. What is the big-$\mathcal{O}$ time for your implementation of `longest_fibm_seq(List)`? For example, is it $\mathcal{O}(1)$, $\mathcal{O}(\log N)$, $\mathcal{O}(N)$, $\mathcal{O}(N \log N)$, $\mathcal{O}(N^2)$, $\mathcal{O}(N^{7/2})$ or something else? Give a short justification for your answer.

   (c) (5 points) Run-time measurement. ← Q3.c
   Use time_it:t to measure the run-time of your implementation of `longest_fibm_seq`. Write a function called `longest_fibm_seq_time()` that for each test case prints a line of the form:

   `N = `*SomeInteger*`, T = `*SomeFloat*

   Where `N` is the length of the list for that timing measurement, and `T` is the mean run-time in seconds reported by `time_it:t`. For your experiments, let `M = 1000000000`. You should have at least eight data points, and they should span a range from small values of N (i.e. $0 \leq N < 10$)

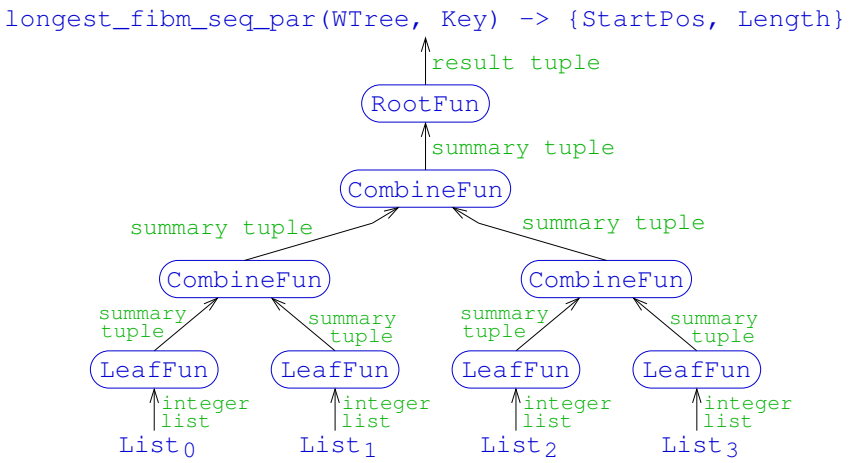

Figure 1: Using reduce for a parallel implementation of "find the longest `M`-Fibonacci sequence"

to a value large enough to obtain a mean execution time T with $0.05 \leq\leq 0.2$. Please report run-times for execution of your code on the machine `thetis.ugrad.cs.ubc.ca`.

Are your measurements consistent with your analysis from part (b)?

Making `longest_fib_list` parallel

Our goal is to write a parallel implementation of `longest_fibm_seq`, we'll call it

```
longest_fibm_seq_par(WTree, Key) -> {StartPos, Length}.
```

where `WTree` is a tree of worker processes created by wtree:create, and `Key` is the key for the list of data in the worker process states. Our plan is to use reduce – figure 1 sketches this approach.

Typically, there are two main parts to solving a problem using reduce:

- Figuring out the operation that will combine results from the left and right subtrees of a node to get the result for their combination.
- Figuring out the data representation for the combine operation.

These two issues are closely connected. I'll start with the data representation, because this is where programmers often get stuck.

From figure 1, we can see that there are *three* main data-types that are used during a reduce:

- the type of the list (or array) elements;
- the type of the summaries that are used by combine;
- the type of the result.

For simple problems, all three of these are the same. For example, if we want to find the sum of the elements of a list or the maximum element in a list, all three are numbers (e.g. integers or floats). The count 3s problem looks like all three are numbers as well, but this slightly deceptive. Consider the "count `bananas`"

problem – give a list, how many elements are the Erlang atom `'banana'`. Clearly, we can solve this with code that is almost identical to the solution for count 3s, but now the elements are Erlang atoms or perhaps arbitrary Erlang terms. For the "second largest" problem, the elements and final result were numbers, but the summaries for combine were tuples of two numbers.
In general, we have the problem of:

> *Given the types for the elements and the result, what type should the arguments and result of the combine operation be?*

It's tempting to try the same type as the result, but sometimes this lacks critical information. Once we've identified what information is needed for the combine operation, we can define a suitable data type. Once we know the data type, writing the actual code for the leaf, combine, and root functions is often straightforward.

The remainder of this homework set addresses these design issues in the following manner:

Question 4 looks at how a list can be distributed across multiple processes. Often, looking at a few examples is a good way to see the pattern. Don't be afraid to try simple examples – there is no extra credit for trying to derive everything from abstract principles!

Question 5 describes the summary tuple that we will use for combine operations. As described above, finding the "right type" for the operands and result of combine is often the hardest part of the problem. For this homework, I'll guide you step-by-step with how we can find such a type. For later problems, we'll ask you to do it yourself.

Question 6 has you write the leaf function.

Question 7 has you write the combine function.

Question 8 has you write the root function. This completes `longest_fibm_seq_par`. We will have you test it.

Question 9 has you measure the runtime of `longest_fibm_seq_par` and compare it with the sequential version.

4. **Distributing the list.** (12 points)
   We will assume that our data is already partitioned across `N_Proc` processes. Each process has a segment of the complete list; these segments are non-overlapping, and they cover the entire list. For example, if the full list is

   ```
   List10 = [1, 2, 4, 6, 10, 15, 25, 35, 8, 16]
   ```

   we could distribute `List10` across four process where

   - Process 0 has segment `[1,2]`;
   - Process 1 has segment `[4,6,10]`.
   - Process 2 has segment `[15,25]`;
   - Process 3 has segment `[35,8,16]`.

   For this example, the longest `M`-Fibonacci sequence is `[2,4,6,10]`. It has one element on process 0, and three elements on process 1.

   Each of the questions below ask you to give an example of a list that is distributed across four processes such that the longest `M`-Fibonacci sequence has the properties specified in the particular question. You should write down the full list, and show what segment is assigned to each process. These segments do not need to be of the same length. You should give the starting position and length of the longest `M`-Fibonacci sequence in the full list.

(a) (3 points) Show an example of a list that is distributed across four processes such that the longest M-Fibonacci sequence for the entire list is contained entirely in the segment for process 1. ← Q4.a

(b) (3 points) Show an example of a list that is distributed across four processes such that the full list has at least two M-Fibonacci sequences of length four and that the first of these M-Fibonacci sequences has one element on process 0 and three elements on process 1. ← Q4.b

(c) (3 points) Show an example of a list that is distributed across four processes such that the longest M-Fibonacci sequence has two elements on one process and three elements on another process. ← Q4.c

(d) (3 points) Show an example of a list that is distributed across four processes such that the longest M-Fibonacci sequence has elements on three different processes. ← Q4.d

5. **Segment summaries** (20 points)
Our subtree summaries will have three components:

- A description of the M-Fibonacci sequence at the left end of the subtree's list.
- A description of the longest M-Fibonacci sequence in the subtree's list.
- A description of the M-Fibonacci sequence at the right end of the subtree's list.

The descriptions of the M-Fibonacci sequences at the left and right ends of the subtree's list are to provide the information that the combine function will need to determine if a longer sequence spans two or more subtrees. These three summaries will be the elements of a list:

```
SubTreeSummary = [Left, Longest, Right].
```

where Left, Longest, and Right are each summary tuples as described below.

I will describe the M-Fibonacci sequence summary that I used in my solution, and prescribe it for your solutions as well. This lets me provide you with some of the more tedious pieces of code so you can focus on the parallel aspects of the problem.

If Segment is a M-Fibonacci sequence, our summary of Segment is an Erlang tuple as shown below:

```
{FirstTwo, {StartPos, Length}, LastTwo}
```

where

- FirstTwo is a list of the first two elements of Segment;
- LastTwo is a list of the last two elements of Segment;
- StartPos is the index of the first element of Segment within the original list.
- Length is the length of Segment.

As an example, let M = 100 and

```
L=[0,1,2,4,6,10,10,20,30,50,80,30,9,48,57,5,62,0]
```

This list L has the (maximal) M-Fibonacci sequences

```
[0,1], [1,2], [2,4,6,10], [10,10,20,30,50,80,30], [30,9], [9,48,57,5,62], [62,0]
```

See figure 2. Note that the last element of one M-Fibonacci sequence of a list is the first element of the next M-Fibonacci sequence of that list. The summary for [10,10,20,30,50,80,30] is

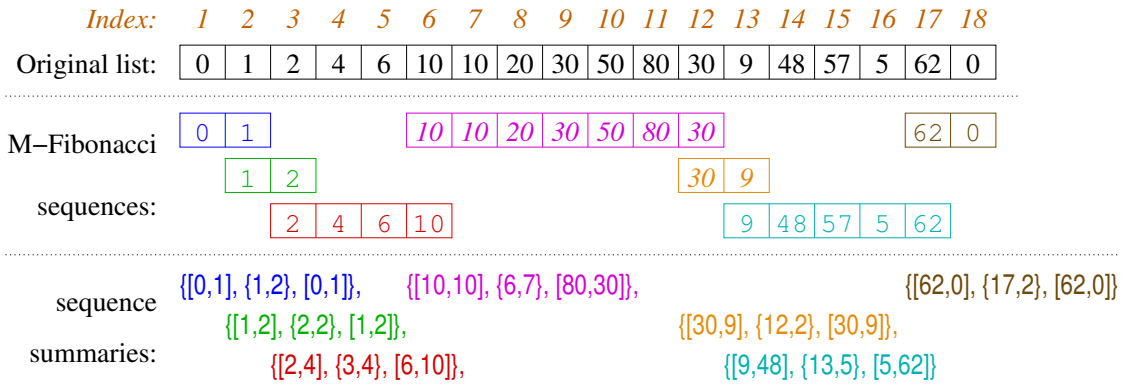

Figure 2: A list and it's M-Fibonacci sequences(M = 100).

```
{[10,10], {6,7}, [80,30]}
```

(a) (5 points) Write the segment summaries for the segments `[0,1]`, and `[9,48,57,5,62]`. ← Q5.a

(b) (5 points) Complete the function `seg_longest/2` from the template. A description of the function and some examples are provided in the template file `hw2.erl`. ← Q5.b

The template code in `hw2.erl` provides accessor functions for segment tuples: `seg_startPos(Segment)`, `seg_length(Segment)`, `seg_lastPos(Segment)`, `seg_prefix(Segment)`, and `seg_suffix(Segment)`. The function `seg_lastPos(Segment)` returns the index within the original list the last element of `Segment`. For example,

```
seg_lastPos({[10,10], {6,7}, [80,30]}) -> 12.
```

Because the final element, `30` is at index 12 in `L`. In the remainder of this question, you will implement a function,

```
fibm_segs(IntegerList, M) -> SegmentList
```

that takes a list of integers as an argument, and returns the corresponding list of segment summary tuples.

(c) (5 points) Write a function, `seg_split(IntegerList, M) -> {FirstSegment, Rest}`, that takes a (possibly empty) list of integers as an argument. Let `Prefix` be the longest prefix of `IntegerList` that is a `M`-Fibonacci sequence. `FirstSegment` is the segment-tuple summary for `Prefix`. If `Prefix` is the entirety of `IntegerList`, then `Rest == []` – there's no more work to be done. Otherwise, the first element of `Rest` is the last element of `Prefix`, and `Rest` is the rest of `IntegerList` – i.e. ← Q5.c

```
IntegerList =:= Prefix ++ tl(Rest).
```

This handles the observation that consecutive `M`-Fibonacci sequences have an overlap of one element.

(d) (5 points) Write a function, `fibm_segs(List, M) -> ListOfSegmentTuples`, that takes a (possibly empty) list of integers as an argument and returns a list of segment summary tuples for the `M`-Fibonacci sequences of `List`. For example (see figure 2), ← Q5.d

```
fibm_segs([0,1,2,4,6,10,10,20,30,50,80,30,9,48,57,5,62,0], 100) ->
  [ {[0,1],{1,2},[0,1]}, {[1,2],{2,2},[1,2]}, {[2,4],{3,4},[6,10]},
    {[10,10],{6,7},[80,30]}, {[30,9],{12,2},[30,9]}, {[9,48],{13,5},[5,62]},
    {[62,0],{17,2},[62,0]}
  ].
```

Hints/notes: In my solution, `fibm_segs(List, M)` is a wrapper function that checks `is_integer_list(List)` and raises an error if `List` is not an integer list. If `List` is an integer list, then my `fibm_segs(List, M)` calls a helper function,

```
fibm_segs(List, StartOffset, M)
```

This function uses `seg_split` (why do you think I had you write it?) to get the first segment summary for `List` and sets the start position for that segment according to `StartOffset`. It then recurses on the rest of the list returned by `seg_split`. My solution is head recursive because I found it easier to think about how to implement my leaf function for the reduce if I didn't reverse the order of the segments.

I included a function `fibm_segs_example` that calls `fibm_segs` for a single test case and prints the result. Feel free to use it for your preliminary testing. Of course, you should include additional tests in your `hw2_tests` module.

6. **The leaf function** (15 points)
Our goal is to write `longest_fibm_seq_par` using wtree:reduce. As you can see from the code template,

```
longest_fibm_seq_par(WTree, Key, M) ->
  wtree:reduce(WTree,
    fun(ProcState) -> fibm_leaf(wtree:get(ProcState, Key), M) end,
    fun(Left, Right) -> fibm_combine(Left, Right, M) end,
    fun(RootTally) ->
      you_need_to_write_this(longest_fibm_seq_par, "root function", [RootTally])
    end
  ).
```

The leaf function for wtree:reduce gets the worker's segment of the full list with wtree:get`(ProcState, Key)`. The value of `M` is visible in the `fun` expression, and the list and codeM are passed to `fibm_leaf`.

(a) (10 points) Write ← Q6.a

```
fibm_leaf(List, M) -> [FirstSeg, LongestSeg, LastSeg]
```

where

- `FirstSeg` is the segment summary tuple for the first segment of `List`;
- `LongestSeg` is the segment summary tuple for the longest segment of `List`; and
- `LastSeg` is the segment summary tuple for the last segment of `List`.

. **Hints/notes:** My solution uses `fibm_segs(List, M)` and `seg_longest(Seg1, Seg2)`. Getting `FirstSeg` from the list returned by `fibm_segs` is straightforward. I wrote a helper function,

```
longest_and_last(SegList) -> {LongestSeg, LastSeg}
```

that returns the segment summary tuple for the longest segment in `SegList` and for the last segment in `SegList`. You can write your own implementation of `longest_and_last`, or you can take another approach if you prefer.

(b) (5 points) Include some test cases for `fibm_leaf` in your `hw2_tests` module. ← Q6.b

7. **The combine function** (30 points)
There are three parts to the combine function, `fibm_combine(Left, Right)`

- The segment summaries for `Left` and `Right` each have `StartPos` values that assume that their lists started at position 1. This if fine for `Left` – we can fix it further up the tree if we need to. For `Right`, we need to add the length of the list for `Left` to the `StartPos` of each segment summary. This is done by the function `seg_shift` which is the topic of question 7a.
- We need to see if the last segment of `Left` and the first segment of `Right` can be merged to create a longer segment. This is the topic of question 7b.
- We need calculate the leftmost and rightmost segments of the combined result. This is the topic of question 7c.

(a) (10 points) Complete the function `seg_shift/2` from the template. A description of the function and some examples are provided in the template file `hw2.erl`. ← Q7.a

Note that the index of the final element of the final segment of `Left` gives you the length of `Left` (bigger hint: `seg_lastPos` is your friend). Try your solution on the examples in the comments – off-by-one errors are a headache, but a few test-cases should help you clean them out.

(b) (10 points) Complete the function `fibm_combine_help/3` from the template. A description of the function and some examples are provided in the template file `hw2.erl`. ← Q7.b

Let the last two elements of `Left` be `[A,B]` (or just `[B]` if `Left` only has one element). Likewise, let the first two elements of `Right` be `[C,D]` (or just `[C]` if `Right` only has one element). Because any list of two integers is a `M`-Fibonacci sequence, `[B,C]` is a `M`-Fibonacci sequence of the list that spans `Left` and `Right`. Because it's not contained in `Left` or `Right`, it hasn't been reported yet. The function `fibm_combine_help` returns the segment summary tuple for the longest `M`-Fibonacci sequence (extending back into `Left` or forward into `Right`) that contains `[B,C]`. Note:

- If `A` is defined and `(A+B) rem M = C`, then the longest `M`-Fibonacci sequence that contains `[B,C]` contains *all* of the last segment of `Left`.
- If `D` is defined and `(B+C) rem M = D`, then the longest `M`-Fibonacci sequence that contains `[B,C]` contains *all* of the first segment of `Right`.

Now, enjoy all the hints and comments that are in the template code.

(c) (10 points) Complete the function `fibm_combine` from the template. A description of the function and some examples are provided in the template file `hw2.erl`. The code first handles the minor cases when the lists for the `Left` and/or `Right` subtrees are empty. For the remaining cases, `fibm_combine` needs to do three things: ← Q7.c

- Shift the `StartPos` for segments from `Right` to account for the length of the list in the left subtree. The template code uses `seg_shift` for this.
- Construct the segment description for the `M`-Fibonacci sequence that includes the last element of `Left` and the first element of `Right`. The template code uses `fibm_combine_help` for this.
- Determine the leftmost segment for the combine of left and right.
Hint: when is the not just the same as `LeftFirst` (see the template code)? If it's not the same as `LeftFirst`, what should it be?
- Determine the longest segment for the combine of left and right.
Hint: what are the possible candidates? Recall `seg_longest` from question 5b.
- Determine the rightmost segment for the combine of left and right.
Hint: this is symmetric with the leftmost problem above.

Yay – now you're done with `fibm_combine`. That's the most challenging part of the code.

8. **Completing `longest_fibm_seq_par`** (10 points)

   (a) (5 points) Complete the function `longest_fibm_seq_par/2` from the template. This part is pretty easy – just fill in the root function. ← Q8.a

   (b) (5 points) Add test cases to your `hw2_tests` module. You are welcome to use the `rand_fibbish_list/3` function from the templates – it produces random lists that have long (according to the `AvgLen` parameter) sublists that are `M`-Fibonacci sequences. You are also welcome to use `fibm_par_test/4` function. It's good for generating timing data (see question 9 below). Your test cases should also include corner cases – typically ones where you run your code with a small number of processes and some short lists to make sure that the various cases in combine are working correctly and other such checks. ← Q8.b

9. **How fast is `longest_fibm_seq_par`?** (15 points)
   Measure the time for `longest_fibm_seq` (the sequential version) and `longest_fibm_seq_par` (the parallel version) for each of the data sets described below. Report the sequential time, the parallel time and the speed-up. Speed-up is defined to be the time to execute the sequential implementation divided by the time to execute the parallel implementation. Use the `fibm_par_test/4` function. Run your experiments on `thetis.ugrad.cs.ubc.ca` a 16-core, 2-way multithreaded machine – that means it can run 32 threads at the same time.

   (a) (5 points) Fix `N_Procs=32`, `M=1000000000`, and `AvgLen=200`. Report the run-times and speed-up for `N_Data` of 100, 1000, 10000, 100000, and 1000000. ← Q9.a

   (b) (5 points) Fix `N_Data=1000000`, `M=1000000000`, and `AvgLen=200`. Report the run-times and speed-up for `N_Procs` of 1, 2, 4, 8, 16, 32, 64, 128, and 256. ← Q9.b

   (c) (5 points) Write some observations about the speed-ups that you observe. Does the data that you observe match the models we presented in class? Where do you see a reasonable correspondance with the models from class? Where do you see discrepancies with those models? ← Q9.c

   Note: the speed-ups for my solution are slower than I had hoped for. I'm guessing that there is substantial overhead in `fibm_seqs`.


Unless otherwise noted or cited, the questions and other material in this homework problem set is copyright 2017 by Mark Greenstreet and are made available under the terms of the Creative Commons Attribution 4.0 International license `http://creativecommons.org/licenses/by/4.0/`