CpSc 418 **Homework 1** Due: Sept. 20, 2017, 11:59pm
Early Bird: Sept. 18, 2017, 11:59pm

65 points (+2 XC because Mark can't add).

Please submit your solution using the handin program. Submit your solution as
cs418 hw1
Your submission should consist of four files:

- `hw1.erl`: Erlang source code for the coding parts your solution.
- `hw1.java`: Java source code for Question 1a.
- `hw1.pdf` or `hw1.txt`: PDF or plain-text for the written response parts of your solution.
- `hw1_test.erl`: EUnit tests for your solution. See question 3.

Templates for `hw1.erl` and `hw1_test.erl` are available at
http://www.ugrad.cs.ubc.ca/~cs418/2017-1/hw/1/code.html.
The tests in `hw1_test.erl` are not exhaustive. If your code doesn't work with these, it will almost certainly have problems with the test cases used for grading. The actual grading will include other test cases as well.

Please submit code that compiles without errors or warnings. If your code does not compile, we might 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.

1. **Fibonacci Numbers** (22 points)
   This question has a **long** question statement. Don't let that scare you. The intention of this question is to bridge the gap between "That seemed so easy when it was presented in lecture." and "I can't figure out how to get started on the homework.". In particular, this question takes you step-by-step through writing a fairly simple Erlang function, and we show many ideas that will be used throughout the course in the process.

   Functional languages make extensive use of recursion. It seems that calculating `N!` and `fib(N)` are obligatory examples when teaching recursion, with `fib(N)` being a popular example because it's easy to write an awful implementation. Here's the standard, awful version of `fib`, written in Erlang:

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

   It is well known that this solution takes exponential time. We can see this using the time_it module from the course Erlang library. For example:

   ```
   1> code:add_path("/home/c/cs418/public_html/resources/erl").
   true.
   2> c(hw1).
   {ok,hw1}
   3> time_it:t(fun() -> hw1:awful_fib(10) end).
   [mean,5.294213176201436e-6,std,5.0459011578069565e-6]
   ```

The call to code:add_path ensures that the course Erlang library is in your Erlang module loading path. I gave the path that works on machines in the `ugrad.cs.ubc.ca` domain. You are welcome to make a copy of the library on your own computer – see the instructions at
`https://www.ugrad.cs.ubc.ca/~cs418/resources/index.html`.

The time_it:t takes a function of no arguments as its parameter. It repeatedly calls the function until it uses about one second of elapsed time. From this, time_it:t calculates the mean and standard deviation of the execution times. In the example above, we see that calls to `awful_fib(10)` took an average of about 5.3 microseconds. One data point doesn't let us conclude anything about the scaling behaviour; so let's try a few more.

```
4> [ [{"N", N} | time_it:t(fun() -> hw1:awful_fib(N) end)] || N <- lists:seq(0,50,5)].
[ [{"N",0},  {mean,9.10e-7}, {std,1.41e-6} ],
  [{"N",5},  {mean,1.24e-6}, {std,5.28e-6}],
  [{"N",10}, {mean,4.32e-6}, {std,1.41e-6}],
  [{"N",15}, {mean,3.73e-5}, {std,7.90e-6}],
  [{"N",20}, {mean,3.94e-4}, {std,6.18e-5}],
  [{"N",25}, {mean,4.36e-3}, {std,5.68e-4}],
  [{"N",30}, {mean,4.78e-2}, {std,2.24e-3}],
  [{"N",35}, {mean,5.37e-1}, {std,6.97e-3}],
  [{"N",40}, {mean,5.97e+0}, {std,undefined}],
  [{"N",45}, {mean,6.58e+1}, {std,undefined}],
  [{"N",50}, {mean,8.11e+2}, {std,undefined}] ]
```

(I did a bit of formatting to make the results easier to read). Does the time to evaluate `awful_fib(N)` appear to be exponential in `N`? For small `N`, not really – we can blame that on overheads in time_it:t function. For $N > 10$, it looks like the time grows by slightly more than a factor of 10 for each increase of `N` by 5. That seems exponential, and we could solve the recurrence and show that the observed growth rate is very close to the predicted one. If you're happy doing that kind of math, great. Otherwise, we can just plot it. Figure 1 shows a plot of $\log_{10}(\mathit{Time}(\texttt{awful\_fib(N)}))$ vs. `N`. It looks pretty close to a straight line for $N \geq 10$ as it should be if the time is growing exponentially – if the time is growing exponentially with `N`, then the **log** of the time will grow linearly with `N`.

(a) **Imperative `fib`** (10 points)
The typical "recursion and iteration" lecture goes on to present a more efficient implementation of `fib` in an imperative language.

i. (4 points) Write a simple, but reasonably efficient implementation of `fib` in Java.

ii. (4 points) What is the big-$\mathcal{O}$ time for your implementation? Is it $\mathcal{O}(\texttt{N})$, $\mathcal{O}(\log \texttt{N})$, $\mathcal{O}(\texttt{N}^2)$, $\mathcal{O}(2^{\texttt{N}})$, something else? Give a short, written justification for your answer. Because the time_it module doesn't work with Java, we won't require timing measurements.

iii. (2 points) Run your code. What values do you get for `fib(0)`, `fib(1)`, `fib(2)`, `fib(3)`, `fib(5)`, `fib(10)`, `fib(50)`, and `fib(100)`?

(b) **Better functional `fib`** (12 points)
The problem with `awful_fib` is that each non-leaf call to `fib` makes `two` recursive calls. It's pretty easy to show that the number of leaf-calls to `fib` is `fib(N)`, and that the total number of calls is $2 * \texttt{fib(N)} - 1$. A little more math shows that `fib(N)` grows as $\Phi^{\texttt{N}}$, where $\Phi = (1+\sqrt{5})/2 \approx 1.62$; no wonder `awful_fib` is so slow.

It's also pretty obvious that most of those calls are redundant. For $N > 1$, we note that `fib(N)` depends on `fib(N-1)` and `fib(N-2)`. We can write a function that returns *both* of these values.

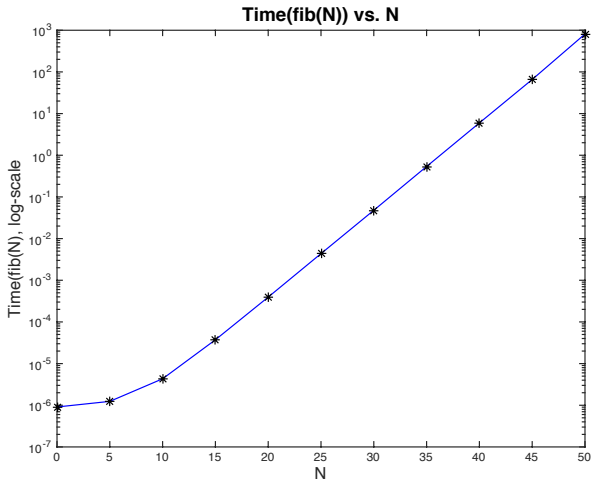

Figure 1: Runtime for `awful_fib`

In particular, for `N` ≥ 1, we will define `fib2` such that

```
fib2(N) -> {fib(N-1), fib(N)}.
```

But, we won't actually call `fib` – instead, you should write the body of `fib2` using a *single* recursive call to `fib2`.

i. (4 points) Write `fib2(N)` as sketched above. Use pattern matching.

ii. (2 points) Write `fib(N)`, again using pattern matching. The main idea is to call `fib2(N)` and extract the return value for `fib(N)` from the tuple returned by `fib2(N)`. Note that `fib(0)` is a special case that can be handled by specifying the appropriate pattern.

iii. (4 points) What is the big-$\mathcal{O}$ time for your implementation? Is it $\mathcal{O}(\mathtt{N})$, $\mathcal{O}(\log \mathtt{N})$, $\mathcal{O}(\mathtt{N}^2)$, $\mathcal{O}(2^{\mathtt{N}})$, something else? Give a short, written justification for your answer. Report timing measurements obtained using time_it:t.

iv. (2 points) Run your code. What values do you get for `fib(0)`, `fib(1)`, `fib(2)`, `fib(3)`, `fib(5)`, `fib(10)`, `fib(50)`, and `fib(100)`?

2. **How fast are some common library functions?** (20 points)

(a) `lists:seq` (5 points) Use `lists:seq` to construct a list, `L1M` with 1,000,000 elements. Use your list `L1M` and time_it:t to measure the time for evaluating `lists:nth(N, L1M)`. Try $N =$ 1, 10, 100, 1000, 10000, 100000, and 1000000. You can try other values for N if you find it helpful. Keep in mind that for small values of N, the measured time is mostly the overhead of the time_it:t function. Does the runtime appear to be $\mathcal{O}(1)$, $\mathcal{O}(\log N)$, $\mathcal{O}(N)$, $\mathcal{O}(N^2)$, $\mathcal{O}(2^N)$, $\mathcal{O}(\mathtt{length(L1M)})$, or something else (please give a formula for "something else")? Include your measurements in your written answer and a short, written justification for your conclusion. Plots are nice but not required.

(b) **length** (5 points)
Measure the time for evaluating length`(List)` for list ranging from 0 elements (i.e. `[]`) to 1,000,000 elements (e.g. `lists:seq(1, 1000000)`). Does the runtime appear to be $\mathcal{O}(1)$, $\mathcal{O}(\log N)$, $\mathcal{O}(N)$,

$\mathcal{O}(N^2)$, $\mathcal{O}(2^N)$, or something else? Include your measurements in your written answer and a short, written justification for your conclusion. Plots are nice but not required.

(c) **element** (5 points)
Use list_to_tuple(L1M) to create a tuple (call it T1M) with 1,000,000 elements. Measure the time for evaluating element(N,T1M) as a function of N. Does the runtime appear to be $\mathcal{O}(1)$, $\mathcal{O}(\log N)$, $\mathcal{O}(N)$, $\mathcal{O}(N^2)$, $\mathcal{O}(2^N)$, $\mathcal{O}$(tuple_size(T1M)), or something else? Include your measurements in your written answer and a short, written justification for your conclusion. Plots are nice but not required.

(d) ++ (5 points)
Let L1 be a list of length N1 and L2 be a list of length N2. Measure the time for evaluating L1 ++ L2 as a function of N1 and N2. Does the runtime appear to be $\mathcal{O}(1)$, $\mathcal{O}(\log$ N1), $\mathcal{O}$(N1), $\mathcal{O}$(N2), $\mathcal{O}$N1 + N2), $\mathcal{O}$(N1 · N2), something else? Include your measurements in your written answer and a short, written justification for your conclusion. Plots are nice but not required.

3. **EUnit tests** (25 points)
EUnit is a unit-testing framework for Erlang. See *The EUnited Nations Council* in *Learn You Some Erlang*. We expect you to write tests for your code. To get you started, we've provided tests for your efficient Erlang implementation of calculating Fibonacci numbers from Question 1b. Here, we will start with a code from the review questions from the Sept. 8 lecture, Basic Erlang:

```
flatten_too_many_ifs(X) ->
  if is_list(X) ->
       if X==[] -> X;
          tl(X) == [] -> X;
          true->
            FlatHead =
            if is_list(hd(X)) -> flatten_too_many_ifs(hd(X));
               true -> hd(X)
            end,
            FlatTail = flatten_too_many_ifs(tl(X)),
            FlatHead ++ FlatTail
       end;
     not is_list(X) ->
       error("flatten_too_many_ifs(X): X is not a list")
  end.
```

As discussed in piazza thread 38, there were two errors in this code.

(a) Write test cases for these bugs (10 points)
Write EUnit tests for flatten_too_many_ifs. This should be in the form of a function called flatten_too_many_ifs_test in hw1_test.erl. You should write at least five test cases including two that catch the known errors in flatten_too_many_ifs.

(b) Complete the review question. (10 points).
Write a function, flatten(X) that *correctly* flattens X. You should use pattern-matching to get a simple implementation. You should correct the errors from flatten_too_many_ifs.

(c) Run your test cases on flatten. (5 points).
It's OK if you do a copy-and-paste to make a flatten_test from flatten_too_many_ifs_test or you can use a test generator that is shared by both, but that's more sophisticated than required for this problem.

Notes: If you find more bugs in `flatten_too_many_ifs` claim the bug-bounties for them. Now that it's a homework problem, they'll get a few more points. The implementation of `flatten_too_many_ifs` is (embarrassingly) inefficient. You don't need to fix the efficiency problem in your solution, but you're welcome to if you want. I might ask to fix the efficiency issues when we devise homework 2.

Why?

Question 1 Fibonacci Numbers:
As stated in the question, this question is a warm-up exercise to get you some experience writing Erlang code. It's a nice example in that it involves using a helper function, `fib2`, using tuples to return multiple values, and it shows you that recursion can be just as fast as iteration.

Question 2 How Fast are Some Library Functions?
This question is to get you more experience making measurements of runtime so you can compare your assumptions with real-world data. The question was inspired by a question on piazza last year: "What is the runtime of `lists:nth`?". The reply (also by anonymous – anonymous seems to talk a lot to himself/herself on piazza) was a link to `http://stackoverflow.com`. OK, you can get it that way, but is the answer correct? It only takes a couple of minutes to run a few tests. That's what this problem is about. The other advantage of getting your own data is that it's up-to-date. If the Erlang runtime has been upgraded since question was answered on stackoverflow, then you can get the wrong answer. If you measure it, you'll know for sure.

Question 3 EUnit tests Testing matters. We provide you with some simple examples for `fib` and you can write some simple tests for `flatten`. It is easy to write tests using EUnit – please get in the habit of testing your code – we'll require it.

Errors, Guards, Efficiency, and Plagiarism

Compiler Errors: if your code doesn't compile, it is likely that you will get a zero on the assignment. Please do not submit code that does not compile successfully. After grading all assignments that compile successfully, we *might* look at some of the ones that don't. This is entirely up to the discretion of the instructors and TAs. If you have half-written code that doesn't compile, please comment it out or delete it.

Compiler Warnings: your code should compile without warnings. In my experience, most of the Erlang compiler warnings point to real problems. For example, if the compiler complains about an unused variable, that often means I made a typo later in the function and referred to the wrong variable, and ended up not using the one I wanted. Of course, the "base case" in recursive function often has unused parameters – use a `_` to mark these as unused. Other warnings such as functions that are defined but not used, the wrong number of arguments to an io:format call, etc., generally point to real mistakes in the code. We will take off points for compiler warnings.

Guards: in general, guards are a good idea. If you use guards, then your code will tend to fail close to the actual error, and that makes debugging easier. Guards also make your intentions and assumptions part of the code. Documenting your assumptions in this way makes it much easier if someone else needs to work with your code, or if you need to work with your code a few months or a few years after you originally wrote it. There are some cases where adding guards would cause the code to run much slower. In those cases, it can be reasonable to use comments instead of guards.

The rest of this discussion of guards is an example showing how a poorly considered guard can change an $O(N)$ time algorithm to $O(N^2)$. It does this by calling `length` in a guard – `length(List)` take time that is linear in the length of `List`. My example for guards that make for terribly slow code is a function `allLess(L1, L2)` that is a check that all elements of `L1` are less than the corresponding elements of `L2`.

```
allLess([], []) -> true; allLess([H1 | T1], [H2 | T2])
    when is_number(H1), is_number(H2),
         is_list(T1), is_list(T2), length(T1) == length(T2) ->
  (H1 < H2) andalso allLess(T1, T2).
```

I tried `allLess(lists:seq(0,999), lists:seq(1,1000))`. Using time_it:t, it takes about 1.4ms (on my laptop) to execute the call to `allLess`. If I change the guard to:

```
when is_number(H1), is_number(H2)
```

Then it takes about 14μs – it's 100 times faster. That's because the function `length` traverses the list and takes time that is linear with the list length. Because the guard is invoked for each recursive call to `allLess`, the guard changes an $O(N)$ computation to $O(N^2)$. In cases like this, it's fine to use the simpler guard. If you call `allLess` with lists of different lengths, this means you'll get an error message showing a call that is different than the one you originally made – it's the call that happened after a bunch of recursive calls, and that can make debugging a little harder – or a lot harder, depending on the details.

A common case for omitting guards occurs with tail-recursive functions. We often write a wrapper function that initializes the "accumulator" and then calls the tail-recursive code. We export the wrapper, but the tail-recursive part is not exported because the user doesn't need to know the details of the tail-recursive implementation. In this case, it makes sense to declare the guards for the wrapper function. If those guarantee the guards for the tail-recursive code, and the tail recursive code can only be called from inside its module, then we can omit the guards for the tail-recursive version. This way, the guards get checked once, but hold for all of the recursive calls. Doing this gives us the robustness of guard checking **and** the speed of tail recursion.

Efficiency: How fast is fast enough? When we are writing parallel code, we are going to the extra effort so we can get performance boost of using many processors in parallel. Efficiency matters. In a bit more detail:

- Big-$\mathcal{O}$ efficiency really matters. As we'll see, parallel processing tends to work best for larger problems. If N is the problem size, and P is the number of processors, then we are looking for cases where $N \gg P$. This means that an extra factor of N in the runtime makes the parallel program *slower* than the sequential one. For example a $\mathcal{O}(N^2/P)$ parallel algorithm is almost certainly slower than a $\mathcal{O}(N)$ sequential algorithm. So, we won't forgive adding an extra factor of N to the runtime of the parallel code.

- This is not a course on advanced, blow-your mind, complicated algorithms. So, if there's an "obvious" $O(N)$ algorithm and a very tricky $O(N \log^* N)$, we'll use the $O(N)$ approach. We probably won't not even mention the $O(N \log^* N)$ approach.

- We'll cover other issues as the course continues. We'll talk a lot about communication overhead. We'll see algorithms where the parallel version has an extra factor of $\log P$ in the number of operations that it performs, and we'll look at why that can be acceptable.

Plagiarism: Please check the collaboration policy at
https://sites.google.com/site/ubccpsc418winter2017term1/assessments.
The short version is that if you use the work of anyone else – another student, a book, something from the web, etc. – and don't cite it, that is academic misconduct. If you provide a clear citation, then it is not academic misconduct. You are welcome and encouraged to discuss homework problems and solution approaches with other students in the class. Please list your collaborators in the solution that you submit. For example: "I discussed question 2 with Jane Foo and Howard Smith." If a specific idea came out of that discussion, please state it. Citing your collaborators will not lower your grade. If we spot similarities in your solutions, the citation will make it clear that it is honest. Of course, we expect each of you to write your own code and do your own analysis whether or not it uses ideas that you got from discussions with other

students. If we ask you to explain a solution that you submitted, you should clearly understand your own work.
Quick review question: is `allLess` tail recursive?

A remark for those who are really into analyzing the code. The behaviour of
```
allLess([1], [0, 0])
```
changes with the guard. The call to `allLess` fails with

```
** exception error:  no function clause matching
  hw1:allLess([1],[0,0]) (hw1.eFootCoderl, line 89)
```

when I use the version with the guards for `T1` and `T2`. When those guards are omitted, `allLess` returns `false`.