CpSc 418 **Homework 1** Due: Jan. 18, 2017, 11:59pm
Early Bird: Jan. 16, 2017, 11:59pm

65 points.

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

- `hw1.erl`: Erlang source code for the coding parts your solution.
- `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 5.

Templates for `hw1.erl` and `hw1_test.erl` are available at
http://www.ugrad.cs.ubc.ca/~cs418/2016-2/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. `closest(P, PointList)`, **(20 points)**.
   The parameter `P` should be a list of numbers. `P` represents a point in a `D=length(P)` dimensional space. The parameter `PointList` should be a list of points in the same space; in other words, each element of `PointList` should be a list of `D` numbers. Find the point in `PointList` that is closest to `P`. `closest` should return a tuple of the form `{Index, Distance}` where `lists:nth(Index, PointList)` is the point in `PointList` that is closest to `Point`, and `Distance` is the distance from `P` to this point. If there are ties, return the index of the *first* element of `PointList` that is closest to `Point`. Write an Erlang implementation of `closest`.

   For your convenience, I have provided a function `distance(P1, P2)` that returns the Euclidean distance from `P1` to `P2`. This is the distance measure you should use in this problem.

2. `allTails(L)`, part 1 **(10 points)**. Let `L` be a list. The function `allTails(L)` returns all suffixes of `L`, including `L` itself. For example, `allTails([1, 2, 3])` should return:
   `[[], [3], [2, 3], [1, 2, 3]]`
   As shown in this example, `allTails` returns the tails from the shortest, `[]`, to the longest, `L`. This makes it natural to write a tail-recursive implementation of `allTails`.

   (a) **(8 points)** Write a tail-recursive implementation of `allTails`.

   (b) **(2 points)** Print the value produced by `allTails(lists:seq(1000, 1010))`. The . . . s to keep the output reasonably short are just fine. A cut-and-paste of the output from the Erlang shell is an ideal answer.

3. `allTails(L)`, part 2 **(10 points)**. The function `erts_debug:size(Term)` returns the number of Erlang "words" used to store *Term*[1]. Let's use this to find out how much space `allTails` uses. Evaluate the Erlang expression below:

[1] `erts_debug` is an "undocumented" module in the Erlang distribution. The function `erts_debug:size` is quite helpful for this problem, but it could be changed or deleted without notice in future Erlang releases.

```
[     {N, erts_debug:size(hw1:allTails(lists:seq(1,N)))}
   || N <- lists:seq(100,1000,100)
]
```

It will generate a list of tuples of the form `{N, SizeTailsN}`, where `SizeTailsN` is the number of memory words needed to store the value from `allTails` when called with a list of `N` integers.

(a) **(5 points)** Does the memory used for `allTails` grow linearly, quadratically, exponentially, or as some other function of `N`? Explain why this is the case, in terms of how Erlang apparently implements the list operations in your `allTails` function. You should explain the "apparently" part – what did you infer from the data?

(b) **(5 points)** If $f(N)$ is linear with N, then a plot of $f(N)$ vs. N will be a straight line. If $f(N)$ is exponential with N, then a plot of $\log(f(N))$ vs. N will be a straight line – you can use $\log_{10}$, ln, or $\log_2$, whichever clearly presents the data. If $f(N)$ is polynomial in N, then for large N, a plot of $\log(f(N))$ vs. $\log(N)$ will be roughly a straight line where the slope of the line is the degree of the polynomial.

Draw a plot, or semi-log plot, or log-log plot of your data to support your answer from part (a). Include a line of best fit. You can use the linear regression functions of Excel, Matlab, R, or your favorite API, or you can just make your own estimate. State how you got your line of best fit.

4. `longestOverlap(L1, L2)` **(20 points)**. The parameters `L1` and `L2` should be lists. We say that `L1` and `L2` *overlap* at positions *N1* through *N2* iff for all *N1* $\leq I \leq$ *N2*
   `lists:nth(`*I*`, L1) =:= lists:nth(`*I*`, L2)`
   Implement the `longestOverlap(L1, L2)` function that returns a tuple, `{Start, Length}` that describes the longest overlapping segments of `L1` and `L2`. In particular, for all `Start` $\leq I <$ `Start`+`Length`
   `lists:nth(`*I*`, L1) =:= lists:nth(`*I*`, L2)`
   "Longest" has the obvious meaning – there can be no tuple `{Start2, Length2}` with `Length2` > `Length` that satisfies the overlap property. If there are two or more longest overlaps, return the `{Start, Length}` with `Start` giving the start position of the first such segment. Finally, if there are no overlaps of `L1` and `L2`, return `{1,0}`, i.e. an overlap of length 0 starting at position 1.

   Here's an example. Let

```
L1 = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20].
L2 = [0,1,2,3,4,6,7,8,10,11,12,13,9,14,15,16,18,19,20].
L3 = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20].
```

   Then,

```
longestOverlap(L1,L2) -> {6, 3} % the sequence 6,7,8
longestOverlap(L1,L3) -> {1, 0} % no overlap
longestOverlap(L2,L3) -> {1, 5} % the sequence 0,1,2,3,4
```

   A few hints. My solution first maps lists `L1` and `L2` to a list, `Match`, where `length(Match) = min(length(L1), length(L2))`. Element *I* of `Match` is true iff the I^{th} elements of `L1` and `L2` match. I used a list comprehension and a function `sloppyZip` that is like lists:zip but doesn't require the lists to be of the same length – it just ignores the leftover elements of the longer list. I have included the code for `sloppyZip` in the template file for `hw1.erl`.

   Second, I wrote a function called `rle(L)`. This function performs run length encoding. Hey! You can get an implementation of `rle` from the solution to HW2 from last year's CPSC 418. Feel free to use that code if you want – of course, you need to cite it. The rest of my solution was an exercise in using functions from the Erlang `lists` module. In particular, my solution uses lists:filter, lists:max, lists:sum,

lists:takewhile, lists:unzip. For that matter, I wrote a new implementation of `rle` using lists:splitwith and then realized there's a version from last year's homework.

Your implementation of `longestOverlap` doesn't have to use these functions or follow the sketch I just wrote. If you find this description helpful, then feel free to use it.

5. **Test cases (5 points):** The test case file distributed with this homework, `hw1_test.erl`, provides a fairly minimal set of EUnit tests. You should definitely add more tests and run them before you submit your solutions to the other problems. Include your version of `hw1_test.erl` with your solution.

   Full credit will be given for any reasonable set of tests. We will also implement a more relaxed collaboration policy *just for* `hw1_test.erl`. If you collaborate with others, you can jointly write one `hw1_test.erl` and all share it. **Of course,** everyone in your collaboration group must submit a copy of the file, and you each must clearly state who your collaborators were in a comment within the first 10 lines of your `hw1_test.erl`, for example:

   ```
   % These test cases were developed jointly by Luke Anakinson, Hans Olo, and Wu Kee.
   ```

Why?

Question 1: This example provides a bit more experience in Erlang programming beyond what was in the first mini assignment. We will use the functions `distance` and `closest` in a later assignment when implementing a parallel version of k-means, a clustering algorithm used in machine learning.

Questions 2, 3, & 4: Finding overlapping DNA sequences is a classic application of parallel computing from bioinformatics. These problems are warm-ups for the string-matching, editing distance, and dynamic programming problems we might get a chance to look at later in the course.

The `allTails` function from questions 2 and 3 provided an opportunity to analyse a bit of the performance of Erlang functions, in this case, their memory usage. We'll be asking you to analyse, measure, and compare throughout the term. These questions are in introduction to that process.

The `longestOverlap` function gave you a chance to write something a bit more involved. We considered of going one more step and combining these three to find the longest common subsequence – that's like `longestOverlap` with out the constraint that the subsequence start at the same position in both lists. This assignment is supposed to be short, and reasonable to do in a week; so, we'll save that for later.

Question 5: Testing is an important part of writing software. Thinking about what the corner cases are, and what the software should do for a wide variety of inputs is essential. This is also an ideal place for collaboration, when you can ask each other "What if . . . ?".

Errors and Guards

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.

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.erl, line 89)
```

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