CpSc 418 **Homework 2** Due: Feb. 3, 2016, 11:59pm
85 points.

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

`hw2.erl` – Erlang source (ASCII text).

`hw2.txt` or `hw2.pdf` – plain, ASCII text or PDF.

The first file, `hw2.erl`, will be your solution to the programming part of the assignment.
Unless otherwise stated, you may use any functions that you like from the Erlang library, for example, functions from the `lists` module.
The second file, `hw2.txt` or `hw2.pdf` should give your solutions to the written parts of the assignment.
No other file formats will be accepted.
A code template is available at hw2.erl.

1. **Getting there faster (15 points + 5 Extra Credit)**
   An intrepid traveler starts out on a journey where the traveler's initial position is described by {*X*, *Y*, *Dir*}, where *X* and *Y* are the traveler's initial x and y coordinates; and *Dir* is the direction that the traveler is facing (in degrees counter-clockwise from the x-axis). The journey is described by a list of moves, where each move has the form {*Turn*, *Dist*}. This tells the traveler to turn *Turn* degrees counter-clockwise, and then go forward *Dist* units of distance. The function `move_pos(`*InitPos*`,` *MoveList*`)` in the code template hw2.erl returns the position and direction of the traveler after completing the moves specified by *MoveList*. Your task, of course, is to do this in parallel.

   (a) **(15 points)**
   Use `wtree:scan` to write a function `move_par(`*W*`,` *InitPos*`,` *MoveKey*`,` *PosKey*`)` where *W* is a worker tree; *InitPos* is the initial position of the traveler; *MoveKey* is the key for a list distributed across the workers describing the journey describing the sequence of moves the traveler is to make; and *PosKey* is the key for storing a list (distributed across the workers) that gives the position of the traveler after each move. `move_par(...)` should return the position and orientation of the traveler at the end of the journey (just like `hw2:mov_pos`), but the computation is done in parallel, and the intermediate positions are stored in the list associated with *PosKey*.

   (b) **(5 points, extra credit)**
   Erlang has some graphics capabilities, see the `wx` package. Use `wx` to draw the path of the traveler.

2. **Run-Length Encoding (10 points)**
   This problem is purely sequential. It's really just a warm-up for the next question. Run-length encoding (RLE) is a simplistic form of data compression. Given a list, RLE replaces each sequence of consecutive, identical elements with a tuple that consists of the value of the element and the number of times it is repeated. For example,

```
rle([a, b, b, b, c, a, a, a, a, z]) ->
   [{a,1}, {b,3}, {c,1}, {a,4}, {z,1}].
```

   Write a function, `rle(List)` that returns the run-length encoded version of `List`.

3. **Longest Run (20 points)**
Given a list, *L*, and a value, *V*, `longest_run(L, V)` is the tuple {*Len*, *Pos*} where *Len* of the longest sequence of consecutive *V*s in *L*, and *Pos* is the position in *L* of the first element in that sequence. For example,

```
longest_run([a, b, b, b, c, a, a, a, a, z], a) -> {4, 6}.
```

Write a parallel version of `longest_run(W, Key, V)` where *W* is a worker tree; *V* is the value we're looking for in the list, and *Key* is the key for the list (distributed across *W*, of course).

Hint: I found it helpful to write a helper function, `rlep`, that's like `rle` described above, and that also includes the position of the start of each run in the tuples.

4. **Sequence Matching (40 points)** There are problems in genomics where a researcher has a genetic sequence, S, and they want to find it's closest match in some genome, G. This problem considers a simplified version of such problems.

Let *L1* and *L2* be lists. We want to find the best amount to shift *L1* relative to *L2* such that the maximum number of elements in *L1* match the corresponding elements in *L2*. Let *Alignment* denote this shift, and *MatchCount* denote the number of matched elements. In the case of ties, we'll take the one with the smallest value for *Alignment*. Here are some examples:

- `L1 = [1,2,3,4]`, `L2 = [0,1,1,2,3,5,8]`, then the `Alignment = 2` and `MatchCount = 3` as shown below:

```
L1 =      [1,2,3,4]          % L1 shifted two places to the right
L2 = [0,1,1,2,3,5,8]
```

- `L1 = [1,2,3,4]`, `L2 = [1,2,3,5,1,5,2,3,4,5]`, then the `Alignment = 0` and `MatchCount = 3` as shown below:

```
L1 = [1,2,3,4]               % L1 unshifted
L2 = [1,2,3,5,1,5,2,3,4,5]
```

- `L1 = [1,2,4,8]`, `L2 = [2,4,6,8,10]`, then the `Alignment = -1` and `MatchCount = 2` as shown below:

```
L1 = [1,2,4,8]               % L1 shifted one place to the left
L2 =   [2,4,6,8,10]
```

(a) **(5 points)** Write a sequential function,

```
best_match(L1, L2) -> {MatchCount, Alignment}
```

that computes `MatchCount` and `Alignment` as described above.

(b) **(10 points)** What is the run-time for your implementation of `best_match` in terms of $N_1 =$ `length(L1)` and $N_2 =$ `length(L2)`. Derive and justify a big-O formula. Measure the run-time of your implementation using the function `time_it:t` in the class Erlang library (i.e. get averages and standard deviation over at least 20 runs for each data point). Compare your empirical measurements with your analytical formula.

(c) **(15 points)** Now write a parallel version:

```
best_match_par(W, Key1, Key2) -> {MatchCount, Alignment}
```

where `W` is a worker tree, and `Key1` and `Key2` are the keys for lists `L1` and `L2` respectively.

(d) **(10 points)** Measure the speed-up (or slow-down) of `best_match_par` compared with `best_match`. For what list sizes is the parallel implementation advantageous? When is the sequential version faster?