Com S 430
Spring 2018
Homework 5

Sample code for problems 2 and 3:
http://web.cs.iastate.edu/~smkautz/cs430s18/homework/hw5/hw5_sample_code.zip

Steve's Erlang notes:
http://web.cs.iastate.edu/~smkautz/cs430s18/examples/erlang/erlang_notes.pdf

Steve's Go notes:
http://web.cs.iastate.edu/~smkautz/cs430s18/examples/go/go_notes.pdf

Some JavaScript examples:
http://web.cs.iastate.edu/~smkautz/cs430s18/examples/js/

Note that most of the examples above just use the JS console for output, so you'll need to have the developer tools open to see anything.

Also remember: to use WebWorkers locally you need to be running a local server. The simplest thing to do is make sure you have Python3 installed, open a command shell, cd to the location of your JS code, and run

```
python -m http.server 2222
```

and then point your browser at localhost:2222.

1) a) Make a simple Javascript "executor", using a WebWorker, that executes an arbitrary function, and define a function **`invokeAsync`** for using it. Once the worker is started, you want to be able to use it like this , where f is a function and args is a JS array containing the arguments.

```
Promise p = invokeAsync(f, args);
```

Unfortunately, you can't send a JS function inside a message to a worker since functions are not cloneable. However, it is easy enough to send it as a string, and then recreate a **`Function`** object from the string. See **`function_test.html`** for an example of doing this. (Note that with this technique, you don't get the closure for the function call, so the function can only refer to local variables declared within the function, and global variables.)

b) Do the Histogram problem in parallel using some configurable number of WebWorkers, with Promises for the results. Use JS typed arrays for the data, which are Transferable. See the example **`test_transferable.html`** .

Put your solutions in in the empty directory p1/hw5 directory of the sample code. There is no particular required structure, just do something reasonable and include comments to make sure it's clear how to test/run it. You don't have to make web pages, console.log output is ok.

2) a) Write a tail-recursive Erlang function that returns the *n*th Fibonacci number. (Put the code in p2/hw5/fib.erl.)

b) Write two Erlang functions which create lists of random integers between 1 and some given upper bound (inclusive).

- randomlists:prepend/2, which prepends new elements at the beginning of the list.
- randomlists:append/2, which appends new elements at the end of the list.

Which is more efficient? Use the given function randomlists:try_it/2 to time the execution of each. For use in the next problem there is also a function randomlists:make/2 which simply delegates to the prepend version, which should be noticeably faster. (This difference points to why the function lists:reverse/1 is so widely used in Erlang.)

All four of the above functions which generate random lists take two arguments. The first parameter is the desired list length; the second is the desired upper bound.

Note that you can generate random integers between 1 and N inclusive with rand:uniform(N). (If you want to reproducibly generate the same list for testing purposes, you can seed the generator using rand:seed/2.)

Also note that you can time the execution of some operations in milliseconds with the following code:

```
Start = erlang:monotonic_time(millisecond),
% Do stuff...
End = erlang:monotonic_time(millisecond),
End - Start.
```

This code is used in the given module stopwatch. Use stopwatch:time_it/1 to time the executions of your two.

c) Write an Erlang function nodups/2 such that if L is a *sorted* list, the call

```
foldl(fun nodups/2, [], L).
```

results in a list containing the same elements as L but with duplicates removed. (I.e., the "accumulator" for the fold is also a list.) If you prefer, the result can be reversed. Make sure that your implementation of nodups/2 works when nodups:try_it/0 is called. Note that try_it/0 uses list:sort/1 to sort a random list for testing.

d) Implement the histogram-calculation example using Erlang processes (a.k.a. actors). The following functions have been added to the histogram module as stubs.

- histogram:make/3, which computes a histogram in parallel.
- histogram:worker/3, which is spawned as part of the above function.
- histogram:make/2, which computes a histogram sequentially from a sorted list
- histogram:from_samples/1, which converts a list of samples to a histogram.
- histogram:merge/2, which combines two histograms.
- histogram:count_samples/1, which counts the total number of samples in a histogram.

Implement each of these functions.

To help you in this process, try implementing your histogram and running histogram:try_it/0, histogram:try_it/3, and the tests in histogram_tests.

You should follow the following basic design.

1. Have the process running histogram:make/3 spawn multiple worker processes, each running histogram:worker/3.
2. Each spawned process runs histogram:make/2 and then sends this histogram back to the process that spawned it via a message send.
3. The main process merges together the histograms as they arrive from the workers.

Recall that all values in Erlang are immutable. Thus, we will not be able to use the same strategy that we used before for generating histogram samples. In particular, we will not be able to modify a mutable array of integers as before.

So, our histograms will be represented with a map-like data structure built on top of Erlang lists. Each element of the list is a mapping from a histogram's bin to the number of samples in that bin.

More specifically, a histogram is a list of 2-tuples, where each 2-tuple is a map entry, {key, value}, relating the bin number to the sample number. So, if the 2-tuple {2, 7} is an element in a histogram list, then this means that the number 2 was a generated sample exactly seven times. These 2-tuples should be ordered by their keys, and no two 2-tuples should have the same key. (For simplicity, we will assume that keys and values should always be integers.)

For example, the histogram

```
1: 5
2: 2
3: 0
4: 0
5: 1
```

is represented in this format as [{1, 5}, {2, 2}, {5, 1}]. Notice that the 2-tuple keys are unique and ordered within the list.

Notes:
(i) All recursion should be tail-recursion.

(ii) For comparison, histogram:try_it(10000000, 10000, 1) on my laptop takes about 12 seconds, and histogram:try_it(10000000, 10000, 8) takes about 2.5 seconds. If your solution runs an order of magnitude longer than this, you probably have a design error.

(ii) In Erlang, we don't have mutable arrays. However, we can we can efficiently "update" the first element of a list by prepending some updated element onto its tail. For example, the following example "updates" L1 when it makes L2 be the same as L1 with its head incremented:

```
15> L1 = [10, 20, 30, 40].
[10,20,30,40]
16> [H|T] = L1.
[10,20,30,40]
17> L2 = [H + 1 | T].
[11,20,30,40]
```

A key point in creating a histogram this way is that the array is sorted, so that you never have to update any tuple other than the head of the list.

3) This problem is written in Go. There are some brief notes referenced at the beginning of this document, which include an intro to the **`go install`** command that you actually use when building and the standard **`go`** directory structure.

You may want to use a Go IDE such as Eclipse and IntelliJ IDEA, or a text editor with Go plugin(s). However, your code will be tested using the **`go`** command line tool. Thus, you may want to make sure that you can compile and run your code with this tool before submitting your assignment.

Your basic task is to implement a solution to the concurrent histogram problem as we have done in previous homework assignments. You can see the required structure in the sample code. You should be able to

- unzip the sample code into some directory foo
- open a command shell and set your GOPATH variable to *path-to-foo*`/p3/hw5`
- cd into `p3/hw5`
- run the command `go install histogram tryit`

The **`histogram`** package contains the histogram code, and **`tryit`** is the main package. So after running **`go install`** as above, you should be able to run **`bin/tryit`**. (It will panic with the message "TODO"). The main function is already implemented, but you will need to edit it to try out your histogram code in different ways.

In this problem, we will represent a histogram with the following struct:

```
type Hist struct {
      Bins []int
      NumSamples int
}
```

The histogram package should include the following functions:

- **`Make`**, which randomly generates a histogram sequentially. It takes the number of samples and an upper bound. It returns a **`Hist`**
- **`MakeInParallel`**, which generates a histogram in parallel. It takes the number of samples, an upper bound, and the number of workers to be used. It returns a **`Hist`**.

Additionally, you should write a **`Merge`** method(see https://gobyexample.com/methods) for the **`Hist`** type such that:

- it merges a given `Hist` instance into a receiver `Hist`.
- the receiver should be a `Hist` *pointer*
- the only parameter is another `Hist`
- the return type should be a boolean `err`, which is set to true if these `Hist` instances' bins are of differing size.
- if upon return `err` is true, then the receiver should not have been changed.

Make sure that your `MakeInParallel()` implementation uses this `Merge` method to merge histograms as they arrive from the workers.