COMP3141 Assignment 1

T.O.R.T.O.I.S.E.

Zoltan A. Kocsis

University of New South Wales
Term 2, 2022

Marking Total of 20 marks (20% of practical component)

Due Date Sunday, 3 July 2022, 23:59.

Submission Instructions The assignment can be submitted using the '`give`' system. To submit from a CSE terminal, type:

```
$ give cs3141 assignment1 Tortoise.hs
```

You will be working in a single file, `Tortoise.hs`. This is the only file you should modify, and the only file you will get to submit.

Version 20 June 2022

Background

Ever since the Aesopian Wars between Tortoise and Hare, and the subsequent development of nuclear weapons, the two species have been locked in a dangerous arms race. By now, each side has stockpiled enough nuclear weapons to destroy the other several times over.

The interspecies community has been trying to negotiate a reduction in nuclear weapons for many years, with little success. While both the Tortoises and the Hares have more

nuclear weapons than they need for deterrence, both sides are afraid of being at a disadvantage if they reduce their nuclear arsenal while the other side doesn't.

Recently, there has been a breakthrough. The two species have finally agreed to sign a treaty (the Treaty On Reducing Threat Of Interspecies Strategic Exchange, or T.O.R.T.O.I.S.E. for short) that will reduce the number of nuclear weapons on each side by half. Unfortunately, the two sides still have their mutual distrust. They fear that the other side will cheat and, instead of dismantling their real nuclear weapons, they will destroy old, useless duds, while secretly keeping their real nuclear stockpiles intact.

To verify that each side is complying with the treaty, they will have to allow inspectors from the other side to analyze the nuclear weapons before they are dismantled. However, the exact layouts and inner workings of such weapons are top secret, and each side is reluctant to allow the others' inspectors to see too much. They agreed that the inspectors will not be permitted to open up and look inside the weapons, but will only use external sensors to measure the radiation signature of each weapon to determine whether it's real or not.

The sensor measurements will be processed to remove any sensitive information, and then compared against a database of known signatures of nuclear weapons. If the signature of a particular weapon does not match anything in the signature database, it means that the weapon is a dud, and not a real nuclear weapon.

You are part of the interspecies task force that has been assigned to write the software underlying the T.O.R.T.O.I.S.E. verification.

Frequencies

In a T.O.R.T.O.I.S.E. test, the inspectors attach a sensor to a nuclear weapon. The sensor measures the presence and strength of ionising radiation, and produces a long list of frequency measurements (these measurements are also called detection events). Every single frequency measurement yields a **positive** integer, represented by the Haskell data type

```
data Freq = Freq Int deriving (Show, Eq, Ord)
```

Frequency Intervals

The results of these frequency measurements have to be compared to a database of known radiation signatures of real weapons. However, measurements are always subject to some random noise and will never match the signature exactly.

To account for this, the signatures are given as **histograms**: tables that tell the expected number of detection events in each **frequency interval**.

In our histograms, every frequency interval has a width of 100 frequency units, and the *start point* of each interval is an integer multiple of 100 frequency units. Frequency intervals are represented by the Haskell data type

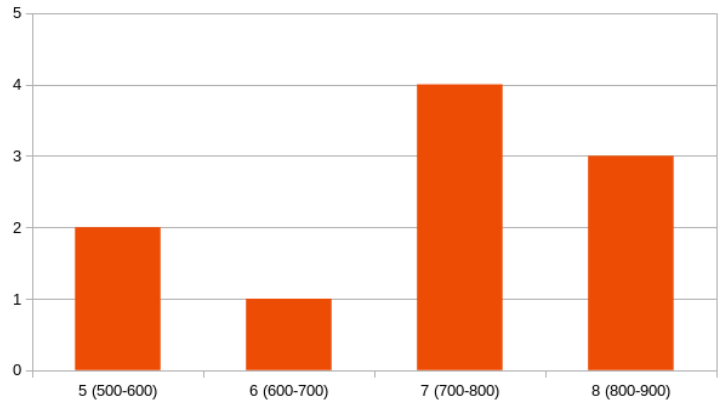
```
data Interval = Interval Int deriving (Eq, Ord)
```

In the spirit of Yaron Minsky's adage, "make illegal states unrepresentable", the value `Interval n` represents the frequency interval starting at $100n$ frequency units, and ending at $100n + 100$ frequency units. For example, `Interval 6` denotes the frequency interval consisting of frequencies between 600 and 700 frequency units.

Moreover, frequency intervals never contain their endpoints: for example, `Interval 3` contains each of the frequencies `Freq 300`, `Freq 301`, `Freq 322`, `Freq 398`, `Freq 399` but it does not contain the frequency `Freq 400`. This is meant to ensure that no frequency is contained in multiple different intervals.

Histograms

Histograms summarize the number of detection events in each frequency interval. As an example, consider the following set of ten detection events: `841`, `739`, `742`, `708`, `863`, `707`, `854`, `586`, `665`, `590`. These detection events can be summarized using the histogram below.



From the histogram, we can read off that two of the events fell into `Interval 5` (the frequency interval consisting of frequencies between 500 and 600 units), while four events fell into `Interval 7`.

The Haskell type declarations

```
type Count = Integer
data Histogram = Histogram [(Interval, Count)] deriving (Show, Eq)
```

are used to define histograms. We say that a given `Histogram` value is *well-formed* if all of the following hold for it:

1. Every `Interval` occurs at most once in the list.
2. Every `Count` that occurs in the list is positive (non-zero).
3. The `Interval`s in the list occur in ascending order.

This is meant to ensure that every histogram has a unique representation. For example, the histogram above can only be represented as

```
Histogram [(Interval 5, 2), (Interval 6, 1), (Interval 7, 4), (Interval 8, 3)]
```

Problem 1: Frequencies and Intervals (5 marks)

a. Implement a function `inside :: Freq -> Interval -> Bool` that tests whether a given frequency measurement belongs to the given frequency interval.

b. Reimplement the appropriate type class instance that allows us to turn `Interval` structures into `String`s, in such a way that the value `Interval 4` will be printed as `"400 to 500"`, the value `Interval 6` will be printed as `"600 to 700"`, and so on.

c. According to the specification above, every frequency should belong to precisely one frequency interval. To ensure that this is the case, implement

1. A function `intervalOf :: Freq -> Interval` which, when given a frequency, returns the unique frequency interval which contains the given frequency.
2. A property `prop_inIntervalOf :: Freq -> Bool` which tests that the interval returned by `intervalOf` indeed contains the given frequency.
3. A property `prop_inOneInterval :: Freq -> Interval -> Property` which tests that the interval returned by `intervalOf` is the *only* frequency interval that contains the given frequency.

To test these properties using QuickCheck, you will have to implement one or more instances of the `Arbitrary` type class.

HINT: By default, QuickCheck's input generation heavily favors small integer values. When generating frequency measurements, you might want to use the `Large Int` modifier to get better test coverage. Using the `verboseCheck` function in place of `quickCheck` will allow you to inspect the generated values: realistic frequency values are between 0 and 2000 units.

Remark: It's okay if `prop_inOneInterval` cannot be run with your current `Freq` generator: the testing scripts will use a better generator to run it.

Marking Criteria	
Marks	Description
1	correctly implementing `inside`
1	correctly implementing pretty-printing of `Interval`
1	correctly implementing `intervalOf`
1	correct testing: input in at least one interval
1	correct testing: input in at most one interval
5	**Total**

Problem 2: Constructing Histograms (4 marks)

a. Following *parse, don't validate*, write a *smart constructor* function

```
histogram :: [(Interval, Count)] -> Histogram
```

for the `Histogram` data type, which returns a histogram value that is well-formed according to the three well-formedness rules given above. If the `Interval`s in the input list do not occur in ascending order, your smart constructor should have them sorted in the returned `Histogram`. If any of the `Count`s in the list are zero or negative, your smart constructor should not include the corresponding `Interval`s in the returned `Histogram`. Every `Interval` that occurs exactly once in the input list with a positive count should also occur exactly once in the returned `Histogram`, with the same corresponding count. Finally, if any of the `Interval`s occur more than once in the input list with a positive count, your smart constructor should include only one of the occurrences in the returned `Histogram` (you can freely choose which one to include).

b. Use the `histogram` function to implement an `Arbitrary Histogram` instance. Write three properties, `prop_histogram1/2/3 :: Histogram -> Bool` which can be used with QuickCheck to verify that your `histogram` function returns well-formed `Histogram` representations (i.e. that `Histogram`s constructed via `histogram` really satisfy the three rules given in the specification above).

Marking Criteria	
Marks	Description
1	every key occurs once in output of `histogram`
1	every count is positive in output of `histogram`
1	duplicate keys are handled correctly in `histogram`
1	the three tests are implemented correctly
4	**Total**

Problem 3: Processing Measurements (5 marks)

a. In a T.O.R.T.O.I.S.E. test, the sensor attached to the nuclear weapon produces a long list of frequency measurements (detection events), represented in Haskell as a `[Freq]`.

The distribution of these events has to be compared to the signatures of known nuclear weapons. To do this, we first have to process the list of detection events into a `Histogram`. Implement a function

```
process :: [Freq] -> Histogram
```

which summarizes a given list of detection events in a histogram showing the number of detection events that occurred in each frequency interval.

For example, processing the list of frequency measurements

```
[Freq 512, Freq 588, Freq 756]
```

on the Haskell REPL should print the following:

```
Histogram [(500 to 600,2),(700 to 800,1)]
```

b. Testing a nuclear weapon can take many days, so the inspectors prefer to do their measurements in multiple shorter batches. These lists of measurements are processed separately, yielding multiple `Histograms`. Before we can compare these to the known weapon signatures, the results first have to be merged (plotted in a single `Histogram`). Merging the separately processed histograms should result in the same final histogram that we would have obtained by processing all the measurements in the same batch. Implement an operation

```
merge :: Histogram -> Histogram -> Histogram
```

which satisfies this property, i.e. one which satisfies the equation

```
merge (process xs) (process ys) = process (xs ++ ys)
```

for all possible lists of detection events `xs,ys`.

Implement `Semigroup` and `Monoid` instances for `Histogram` using the operation above. Write three property tests,

```
prop_mergeAssoc :: Histogram -> Histogram -> Histogram -> Bool
prop_mergeComm :: Histogram -> Histogram -> Bool
prop_mergeId :: Histogram -> Bool
```

which can be used with QuickCheck to verify that the structure consisting of the type `Histogram`, the binary operation `merge`, and the appropriate unit histogram together form a commutative monoid.

Marking Criteria	
Marks	Description
1	correctly implementing `process`
1	`merge` satisfies the expected equation
1	correctly defining the monoid instance
1	correctly implementing the two monoid tests
1	correctly testing commutativity
5	**Total**

Problem 4: Comparing Histograms (6 marks)

To decide whether a nuclear weapon is real or a dud, the histograms measured by the sensor have to be checked to see if they are similar to the signatures (reference histograms) of known nuclear weapons.

We use the Euclidean metric to compare the similarity/distance of two histograms. Take two histograms g, h. Let g_i denote the count of elements in `Interval i` of the histogram g, and h_i denote the count of elements in `Interval i` of the histogram h.

The Euclidean distance $d(g, h)$ between the two histograms is defined as

$$d(g,h) = \sqrt{\sum_i (g_i - h_i)^2}$$

where the summation index i ranges over all intervals that occur in either of the histograms.

For example, we can calculate the distance between the histograms

```
g = Histogram [(100 to 200, 2)]
h = Histogram [(100 to 200, 5), (200 to 300, 4)]
```

as $d(g,h) = \sqrt{(2-5)^2 + (0-4)^2} = \sqrt{3^2 + 4^2} = \sqrt{9+16} = \sqrt{25} = 5$.

Notice that this is exactly the same as the ordinary geometric distance between the points $(2, 0)$ and $(5, 4)$ in the plane.

a. In a T.O.R.T.O.I.S.E. test, we say that two `Histogram`s are *similar* if their Euclidean distance is less than 32.

A fellow engineer, Inspector O'Hare, claims that similarity of histograms is an *equivalence relation*. Using QuickCheck or otherwise, investigate the claims of Inspector O'Hare. As a report of your results, define three values

```
report_refl :: Maybe Histogram
report_symm :: Maybe (Histogram, Histogram)
report_tran :: Maybe (Histogram, Histogram, Histogram)
```

corresponding to the three usual properties of an equivalence relation. The `report_XXXX` value should be `Nothing` if the corresponding property `XXXX` holds for all histograms. If the corresponding property does not hold for all histograms, then the `report_XXXX` value should be of the form `Just c`, where the value/tuple `c` contains a counterexample to the corresponding property `XXXX` (i.e. the tuple c lists histograms which together violate the claimed property).

b. After the introduction of the T.O.R.T.O.I.S.E. testing program, it was found that sensor measurements in certain frequency intervals were unreliable: detection events in such intervals are subject to too much randomness, and have to be excluded from the calculations when we compute distance/similarity scores between the measured histograms and the known signatures.

The signatures of known nuclear weapons are stored in the form of *signature cards*. A signature card consists of two pieces of data, a references histogram and a list of excluded intervals. The following Haskell data type stores these signature cards.

```
data SigCard = SigCard {
    refHistogram :: Histogram,
    excluded :: [Interval]
  } deriving (Show, Eq)
```

We say that a measurement histogram h *matches* a signature card `SigCard r e` if, after removing the counts for all excluded intervals e from both the measurement histogram h and the reference histogram r, the resulting histograms are similar, i.e. their Euclidean distance is less than 32.

For example, the histogram

```
Histogram [ (100 to 200,14),(200 to 300,5)
          , (900 to 1000,3),(1000 to 1100,13)
          , (1600 to 1700,16)
          ]
```

matches the signature card

```
SigCard {
    refHistogram = Histogram [ (500 to 600,13),(1000 to 1100, 13)
                             , (1100 to 1200,3),(1200 to 1300,12)
                             , (1400 to 1500, 14),(1800 to 1900,8)
                             ],
    excluded = [1600 to 1700,1400 to 1500]
  }
```

since, after excluding the two frequency intervals 1600 to 1700 and 1400 to 1500 from both histograms, the distance between the histograms becomes

$$\sqrt{14^2 + 5^2 + 13^2 + 3^2 + (13 - 13)^2 + 3^2 + 12^2 + 8^2} < 32.$$

Note that, had we not excluded the two intervals, the distance would have slightly exceeded 32.

Inspector O'Hare produced an implementation of the function

```
data Verdict = RealWeapon | Dud deriving (Show, Eq)
match :: Histogram -> SigCard -> Verdict
```

This function is supposed to return the value `RealWeapon` if the given histogram of measurements matches the given signature card, and the value `Dud` otherwise.

However, Inspector O'Hare never took COMP3141, and didn't know how to test the implementation. In fact, the `match` function contains a serious bug, which would allow the other side to design *false positives*: dud weapons that nevertheless pass the `match` process as if they were real weapons.

Using QuickCheck or otherwise, find such a false positive, taking the following simple signature card as your reference histogram:

```
refCard :: SigCard
refCard = SigCard r v where
  r = Histogram [(Interval 4, 4000), (Interval 5, 6000), (Interval 6,300)]
  v = [Interval 5]
```

Define your false positive as the value

```
falsePos :: Histogram
```

You found a correct false positive if evaluating `match falsePos refCard` returns `RealWeapon`, even though a correct implementation of the same function would return `Dud`.

Hint: This is a difficult problem! You will have to use QuickCheck creatively. Try starting from the reference histogram!

Marking Criteria	
Marks	Description
1	correctly implementing `report_refl`
1	correctly implementing `report_symm`
1	correctly implementing `report_tran`
3	finding a false positive input
6	**Total**

End of Problem Statements.

Overview

This assessment is intended to give you experience writing Haskell programs using functional programming idioms, as well as experience reading, writing and complying with mathematical specifications and QuickCheck properties. It also tests your knowledge of common type classes, including `Semigroup`, `Monoid`, `Show`.

You have almost two weeks to complete the assignment, but beware: a full solution will involve writing approximately 120 lines of Haskell code, and doing quite a bit of thinking. Start early!

Historical Background. The scenario presented in this assignment is inspired by real world events. S.T.A.R.T. (the Strategic Arms Reduction Treaty) was an agreement between the United States of America and the Soviet Union, aimed at reducing and limiting the number of strategic nuclear weapons stockpiled by the two nations. The treaty was signed on 31 July 1991 and entered into force on 5 December 1994. S.T.A.R.T. was the largest and most complex arms control treaty in history, and its final implementation in resulted in the dismantling of about 80% of all strategic nuclear weapons then in existence. Implementing arms reduction treaties requires software engineers to solve problems that are similar to (but much more complicated than) the ones presented in the assignment.

Marking and Testing

All marks for this assignment are awarded based on *automatic marking scripts*. Marks are not awarded subjectively, and are allocated according to the marking criteria presented at the end of each problem statement. The scripts that run when you submit the assignment are similar to (but not identical with) the scripts that will be used to determine your final marks: you are advised to do your own testing, instead of relying solely on the submission test suite.

Barring exceptional circumstances, the marks awarded by the automatic marking script are *final*. For this reason, plese make sure that your submission compiles and runs correctly on CSE machines, and that the submission scripts do not report any problems.

Late Submissions

Unless otherwise stated if you wish to submit an assignment late, you may do so, but a late penalty reducing the maximum available mark applies to every late assignment. The maximum available mark is reduced by 5% if the assignment is one day late, 10% if the assignment is two days late, 15% if the assignment is three days late. Assignments that are late 4 days or more will be awarded zero marks.

Extensions

Assignment extensions are only awarded for serious and unforeseeable events. Having the flu for a few days, deleting your assignment by mistake, going on holiday, work commitments, etc do not qualify. Aim to complete your assignments well before the due date in case of last minute illness, and make regular backups of your work.

Plagiarism

All work submitted for assessment must be **entirely your own**. Unacknowledged copying of material, in whole or part, is a serious offence. Before submitting any work you should read and understand the UNSW Plagiarism Policy.

In this course submission of any work derived from that of another person, or solely or jointly written by or with someone else, without clear and explicit acknowledgement, will be severely punished, including with automatic failure and an overall mark of zero for the course. This includes using unreferenced work taken from books, web sites, etc.

Do not share your work with any other person! Allowing another student to copy you work will, at the very least, result in zero for that assessment. If you knowingly provide or show your work on this assignment to another person for any reason,, and work derived from it is subsequently submitted, you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep all your work private. If you are unsure about whether certain activities constitute plagiarism, you should ask us before engaging in them!