PLC: Project 1 [180 points]

Due date: Progress report due March 4th, by midnight; solution due March 11th, by midnight

About This Project

You will implement a tool for working with *sorting networks*. We will talk briefly about these in class Feb. 22nd. For a nice introduction, see

`https://hoytech.github.io/sorting-networks/`.

A very solid reference is this:

`http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap28.htm`.

This is a chapter from an online copy of Cormen, Leiserson, Rivest (CLR), and it is excellent. I will reference it in the instructions below. Note that the first resource seems to be sorting elements with smaller numbers towards the bottom of the circuit, while the second one puts smaller numbers towards the top. I will go with the second.

Important: You must submit a progress report (by adding it to your Github Classrooms repo) called `progress.txt` by March 4th, midnight. This is described more below.

Since this is a project rather than a homework, I am just providing you a specification, but no starting code. You must implement your solution in Haskell, of course.

How to Turn In Your Solution

Follow this link to create your repo on Github Classrooms for project 1:

`https://classroom.github.com/a/qHLDFN6E`

If you work with a partner, please add to your repo a file called `partners.txt`, and list your name and your partner's name there. Both of you should submit your solution: each partner pushes it to their github repo.

How To Get Help

You can post questions in the Discussions section of ICON.

You are also welcome to come to our office hours. See the course's Google Calendar, linked from the github page for the class, for times and Zoom links for office hours. You can also find these on the "Zoom links, office hours" page under "Pages" on ICON:

`https://uiowa.instructure.com/courses/179776/pages/office-hours`

1 Progress report [30 points]

By March 4th, midnight, you need to add and push a plain text file called `progress.txt` to your project's repo. In this file, please briefly describe your progress on the problems below, and any difficulties you are having.

`Main.hs` and command-line arguments

In this assignment, unlike previous ones, the goal is to create an executable program that can be invoked from a terminal using `runhaskell Main.hs`. So you need to create a file `Main.hs` holding the `main :: IO ()` function for your program. You may create whichever other files are helpful for organizing your solution to the problems below. Don't forget to add, commit, and push all these files to your repo on github.

For each of the five parts below (numbered 2 through 6), I describe the command-line arguments that trigger the operation you are supposed to implement for that part. For an example of how to read command-line arguments in Haskell, see the `Main.hs` file from week 5. If your program is invoked with unrecognized command-line arguments, then it should print an error message.

Each part is worth 30 points. Some parts are likely harder than others, even though the points are the same for each. The parts are somewhat independent, although you need to be able to read comparator networks (part 2) for all but part 6. You also need to be able to run comparator networks (part 3) to test if such a network is a sorting network (part 5). Otherwise the problems are independent, so you can try them in whichever order you like.

Error handling is of great importance in software, but to keep the project manageable I have not demanded that you provide nice error messages if things are not in the expected format.

Testing your code on other inputs besides `sort1.txt` (provided) is advised, as we will use private tests as part of grading. To do this, we are including an encrypted private test, `comp1.txt.cpt`, which will be decrypted and run after the submission deadline. (Please do not remove or modify this file.)

2 Reading and writing comparator networks [30 points]

A *comparator network* can be represented as a list of pairs of numbers. Each pair (x, y) represents a comparator acting on the two wires numbered x and y. If the value on wire y is smaller than that on wire x, then the comparator swaps the values of the wires. One executes a network by propagating an initial set of values on the wires through the various comparisons. The resulting values might or might not be sorted.

An example is in `sort1.txt`:

```
[(1,2),(3,4),(1,3),(2,4),(2,3)]
```

This is the network in Figure 28.2 of the CLR chapter cited above.

For this first part, your program needs to read a command-line that looks like

```
Read filename
```

where `filename` is the name of a file expected to contain a sorting network. For simplicity, it is ok if your program fails with a run-time exception if the file is not in the required format (of a list of pairs of integers), or does not exist. You should read in the network and then print it out again to a file called `network.txt` in a format where each comparison is on its own line, like this (for the network in `sort1.txt`):

```
1 -- 2
3 -- 4
1 -- 3
2 -- 4
2 -- 3
```

The comparisons should appear in the same order as in the input.

There is a test to check that the output for `sort1.txt` matches the example output in the file `network1.txt`. Please try to match the spacing (like one space on each side of the "–"), for autograding purposes.

3 Running a comparator network [30 points]

For this part, you have to implement running a comparator network on an input sequence. The command-line for this is

```
Run filename sequence
```

where `filename` is the name for a file containing a comparator network (like `sort1.txt`), and `sequence` is a list like `[5,1,3,0]`. You may assume that the list is in the correct format, that all numbers in the list are distinct, and that its length is greater than or equal to the biggest wire number in the comparator network that will be used. (If the list is longer, that just means that a suffix of it will not be affected when it is run through the comparator network.)

You should print out the result of applying the network to the sequence. Just print it to the standard-output channel of the program using a function like `putStrLn`, and print it out in the Haskell format for lists of integers (you can just call `show` on a list you compute in your code).

There is a test to check that the running `sort1.txt` on `[5,1,3,0]` produces `[0,1,3,5]`.

4 Putting comparator networks into parallel form [30 points]

In a comparison network, comparisons (x, y) and (a, b) may be done in parallel when the set $\{x, y\}$ is disjoint from $\{a, b\}$. In that case, the two comparisons are working on separate pairs of wires, so they could execute simultaneously.

We can represent a comparator network in **parallel form** as a list of lists. Each element list shows comparisons that can happen in parallel. For example, a parallel form for `sort1.txt` is the list of lists:

```
[[(1,2),(3,4)],[(1,3),(2,4)],[(2,3)]]
```

The command line for this part is

```
Parallel filename
```

Here, `filename` is again the name of a file containing a comparator network, like `sort1.txt`. Your code should write the parallel form to a file `parallel.txt`, in this format (for the above example):

```
1 -- 2 , 3 -- 4
1 -- 3 , 2 -- 4
2 -- 3
```

On each line, please order the comparators by first wire number. So it would not be legal to have

```
3 -- 4 , 1 -- 2
```

because the first wire number (3) of the first comparison is greater than the first wire number (1) of the second. Please use the same spacing as above so that your output can be easily compared with expected output.

Note that I am not giving you an algorithm for finding the parallel form! Please come up with one yourself. Alternatively, if you manage to find an algorithm online, that is fine, too, but please just credit your source (for example, in your code). Parallel forms might not be unique for some networks. You will receive full credit if you compute any correct parallel form. The form you compute should be maximal, though, in the sense that if you have two lists of comparators next to each other in the parallel form, then it is not possible to merge those into a single list of parallel comparators.

There is a test to check that the parallel form you produce for `sort1.txt` matches `parallel1.txt`. (Please try to get it to match exactly, but if it is the same except for spacing or other issues, you will probably get full credit.)

5 Testing if a comparator network is a sorting network [30 points]

A *sorting network* is a comparator network where the final values on the wires are indeed always sorted (smaller numbers towards the top), no matter what initial values we start with.

The zero-one principle discussed in the CLR chapter states that it is enough to make sure a comparator network with n wires always correct sorts lists of 0s and 1s. The more general case, where it sorts lists of numbers 1 through n, follows.

The command-line for this part is

```
Sorting filename
```

Print out `True` if the comparator network in the given file is a sorting network, and `False` otherwise. (So you have to write code to test this property.)

There is a test to check that your program prints `True` for `sort1.txt`.

6 Computing a simple sorting network [30 points]

Figure 28.3 of the CLR chapter shows an example sorting network, based on insertion sort. Given a number $n \geq 0$ of wires, for this part you should compute the network similar to that example (but for that number of wires). The command line is

```
Create n
```

where n is the number of wires. You should write your network to `parallel.txt`, in parallel form (part 4 above).

There is a test to check that the network your program produces when n is 5 matches the one in `insertion5.txt`.