COMP226 Assignment 1: Reconstruct a Limit Order Book

Continuous Assessment Number	1 (of 2)
Weighting	10%
Assignment Circulated	09:00 Tuesday 18 February 2020 (updated 2020-02-20)
Deadline	17:00 Friday 6 March 2020
Submission Mode	Electronic only http://www.csc.liv.ac.uk/cgi-bin/submit.pl Submit a single file "MWS-username.R", where MWS-username should be replaced with your MWS username.
Learning Outcomes Assessed	Have an understanding of market microstructure and its impact on trading.
Goal of Assignment	Reconstruct a limit order book from order messages
Marking Criteria	Code correctness (85%); Code readability (15%)
Submission necessary in order to satisfy module requirements	No
Late Submission Penalty	Standard UoL policy; **resubmissions after the deadline will NOT be considered**.
Expected time taken	Roughly 8-12 hours

Warning

Your code will be put through the department's automatic plagiarism and collusion detection system. Student's found to have plagiarized or colluded will likely receive a mark of zero. Do not discuss or show your work to others. In previous years, two students had their studies terminated and left without a degree because of plagiarism.

Rscript from Rstudio

In this assigment, we use `Rscript` (which is provided by R) to run our code, e.g.,

```
Rscript skeleton.R input/book_1.csv input/empty.txt
```

In R studio, you can call Rscript from the "terminal" tab (as opposed to the "console"). On Windows, use Rscript.exe not Rscript:

```
Rscript.exe skeleton.R input/book_1.csv input/empty.txt
```

Distributed code and sample input and output data

As a first step, please download ~~comp226_a1.zip~~ `comp226_a1_v3.zip` from:

https://student.csc.liv.ac.uk/internal/modules/comp226/_downloads/comp226_a1_v3.zip

Then unzip `comp226_a1.zip`, which will yield the following contents in the directory `comp226_a1`:

```
comp226_a1
├── input
│   ├── book_1.csv
│   ├── book_2.csv
│   ├── book_3.csv
│   ├── empty.txt
│   ├── message_a.txt
│   ├── message_ar.txt
│   ├── message_arc.txt
│   ├── message_ex_add.txt
│   ├── message_ex_cross.txt
│   ├── message_ex_reduce.txt
│   └── message_ex_same_price.txt
├── output
│   ├── book_1-message_a.out
│   ├── book_1-message_ar.out
│   ├── book_1-message_arc.out
│   ├── book_2-message_a.out
│   ├── book_2-message_ar.out
│   ├── book_2-message_arc.out
│   ├── book_3-message_a.out
│   ├── book_3-message_ar.out
│   └── book_3-message_arc.out
└── skeleton.R

2 directories, 21 files
```

Brief summary

The starting point for the assignment is a code skeleton, provided in a file called `skeleton.R`. This file runs without error, but does not produce the desired output because it contains 6 empty functions. To complete the assignment you will need to correctly complete these 6 functions.

You should submit a single R file that contains your implementation of some or ideally all of these 6 functions. Your submission will be marked via a combination of:

- automated tests (for **code correctness, 85%**, breakdown by function given below); and
- human visual inspection (for **code readability, 15%**, in particular, for appropriate naming of variables and functions (5%), good use of comments (5%), and sensible, consistent code formatting (5%)).

Correct sample output is provided so that you can check whether your code implemetations produces the correct output.

skeleton.R versus solution.R

You are given skeleton.R, which you should extend by implementing 6 functions. Throughout this handout, we also generate example output using a file solution.R that contains a correct implementation of all 6 of these functions. Obviously, you are not given the file solution.R, however the example output will be helpful for checking that your function implementations work correctly.

Two sets of functions to implement

As described in detail in the rest of this document, you are required to implement the following 6 functions. The percentage in square brackets correspond to the breakdown of the correctness marks by function.

Limit order book stats:

1. `book.total_volume <- function(book)` **[10%]**
2. `book.best_prices <- function(book)` **[10%]**
3. `book.midprice <- function(book)` **[10%]**
4. `book.spread <- function(book)` **[10%]**

Updating the limit order book:

5. `book.reduce <- function(book, message)` **[15%]**
6. `book.add <- function(book, message)` **[30%]**

Warning

Do not make changes to the rest of the code in `skeleton.R`, only implement these 6 functions. Penalties may be applied if other changes are present in your submission.

Running skeleton.R

An example of calling skeleton.R follows.

```
Rscript skeleton.R input/book_1.csv input/empty.txt
```

As seen in this example, skeleton.R takes as arguments the path to **two input files**:

1. initial order book (`input/book_1.csv` in the example)
2. order messages to be processed (`input/empty.txt` in the example)

Note: the order of the arguments matters.

Let's see part of the source code and the output that it produces.

```
if (!interactive()) {
    options(warn=-1)
    args <- commandArgs(trailingOnly = TRUE)

    if (length(args) != 2) {
        stop("Must provide two arguments: <path_to_book> <path_to_messages>")
    }
    book_path <- args[1]; data_path <- args[2]

    if (!file.exists(data_path) || !file.exists(book_path)) {
        stop("File does not exist at path provided.")
    }

    book <- book.load(book_path)
    book <- book.reconstruct(data.load(data_path), init=book)

    book.summarise(book)
}
```

So in short, this part of the code:

- checks that there are two command line arguments
- assigns them to the appropriate variables (the first to the initial book file path, the second to the message file path)
- loads the initial book
- reconstructs the book according to the messages
- prints out the book
- prints out the book stats

Let's see the output for the example above:

```
$ Rscript skeleton.R input/book_1.csv input/empty.txt
$ask
  oid price size
1   a   105  100

$bid
  oid price size
1   b    95  100

Total volume:
Best prices:
Mid-price:
Spread:
```

Now let's see what the output would look like for a correct implementation:

```
$ Rscript solution.R input/book_1.csv input/empty.txt
$ask
  oid price size
1   a   105  100

$bid
  oid price size
1   b    95  100
```

```
Total volume: 100 100
Best prices: 95 105
Mid-price: 100
Spread: 10
```

You will see that now the order book stats have been included in the output, because the four related functions that are empty in `skeleton.R` have been implemented in `solution.R`.

The initial order book

Here is the contents of `input/book_1.csv`, which is one of the 3 provided examples of an initial book:

```
oid,side,price,size
a,S,105,100
b,B,95,100
```

Let's justify the columns to help parse this input:

oid	side	price	size
a	S	105	100
b	B	95	100

The first row is a header row. Every subsequent row contains a limit order, which is described by the following fields:

- `oid` (order id) is stored in the book and used to process (partial) cancellations of orders that arise in "reduce" messages, described below;
- `side` identifies whether this is a bid ('B' for buy) or an ask ('S' for sell);
- `price` and `size` are self-explanatory.

Existing code in skeleton.R will read in a file like `input/book_1.csv` and create the corresponding two (possibly empty) orders book as two data frames that will be stored in the list `book`, a version of which will be passed to all of the six functions that you are required to implement.

Note that if we now change the message file to a non-empty one, `skeleton.R` will produce the same output (since it doesn't parse the messages; you need to write the code, functions 5 and 6, to do that):

```
$ Rscript skeleton.R input/book_1.csv input/message_a.txt
$ask
  oid price size
1   a   105  100

$bid
  oid price size
1   b    95  100

Total volume:
Best prices:
Mid-price:
Spread:
```

If correct message parsing and book updating is implemented, `book` would be updated according to `input/adds_only.txt` to give the following output:

```
$ Rscript solution.R input/book_1.csv input/message_a.txt
$ask
  oid price size
8   a   105  100
7   o   104  292
6   r   102  194
5   k    99   71
4   q    98  166
3   m    98   88
2   j    97  132
1   n    96  375

$bid
  oid price size
1   b    95  100
2   l    95   29
3   p    94   87
4   s    91  102

Total volume: 318 1418
Best prices: 95 96
Mid-price: 95.5
Spread: 1
```

Before we go into details on the message format and reconstructing the order book, let's discuss the first four functions that compute the book stats, which we also see correctly computed in this example.

Computing limit order book stats

The first four of the functions that you need to implement compute limit order book stats, and can be developed and tested without parsing the order messages at all. In particular, you can develop and test the first four functions using an empty message file, `input/empty.txt`, as in the first example above.

The return values of the four functions should be as follows (where as usual in R single numbers are actually numeric vectors of length 1):

- `book.total_volumes` should return a list with two named elements, `bid`, which should contain the total volume in the bid book, and `ask`, which should contain the total volume in the ask book;
- `book.best_prices <- function(book)` should return a list with two named elements, `bid`, which should contain the best bid price, and `ask`, which should contain the best ask price;
- `book.midprice` should the midprice of the book;
- `book.spread` should the spread of the book;

You should check that the output of these functions in the example above that uses `solution.R` are what you expect them to be.

We now move on to the reconstructing the order book from the messages in the input message file.

Reconstructing the order book from messages

You do not need to look into the details of the (fully implemented) functions `book.reconstruct` or `book.handle` that manage the reconstruction the book from the starting initial book according to the messages.

In the next section, we describe that there are two types of message, "Add" messages and "Reduce" messages. All you need to know to complete the assignment is that messages in the input file are processed in order, i.e., line by line, with "Add" messages passed to `book.add` and "Reduce" messages passed to `book.reduce`, along with the current `book` in both cases.

Message Format

The market data log contains one message per line (terminated by a single linefeed character, `'\n'`), and each message is a series of fields separated by spaces.

There are **two types of messages**: "Add" and "Reduce" messages. Here's an example, which contains an "Add" message followed by a "Reduce" message:

```
A c S 97 36
R a 50
```

An "Add" message looks like this:

```
'A' oid side price size
```

- '`A`': fixed string identifying this as an "Add" message;
- `oid`: "order id" used by subsequent "Reduce" messages;
- `side`: '`B`' for a buy order (a bid), and an '`S`' for a sell order (an ask);
- `price`: limit price of this order;
- `size`: size of this order.

A "Reduce" message looks like this:

```
'R' oid size
```

- '`R`': fixed string identifying this as a "Reduce" message;
- `oid`: "order id" identifies the order to be reduced;
- `size`: amount by which to reduce the size of the order (*not* the new size of the order); if `size` is equal to or greater than the existing size of the order, the order is removed from the book.

Processing messages

"Reduce" messages will affect at most one existing limit order in the book.

"Add" messages will either:

- **not cross the spread** and then add a single row to the book (orders at the same price are stored separately to preserve their distinct "oid"s);
- **cross the spread** and in that case can affect any number of orders on the other side of the book (and may or may not result in a remaining limit order for residual volume).

The provided example message files are split into cases that include crosses and those that don't to help you develop your code incrementally and test it on inputs of differing difficulty.

We do an example of each case, one by one. In each example we start from `input/book_1.csv`; we only show this initial book in the first case.

Example of processing a reduce message

```
$ Rscript solution.R input/book_1.csv input/empty.txt
$ask
  oid price size
1   a   105  100

$bid
  oid price size
1   b    95  100

Total volume: 100 100
Best prices: 95 105
Mid-price: 100
Spread: 10
```

```
$ cat input/message_ex_reduce.txt
R a 50
```

```
$ Rscript solution.R input/book_1.csv input/message_ex_reduce.txt
$ask
  oid price size
1   a   105   50

$bid
  oid price size
1   b    95  100

Total volume: 100 50
Best prices: 95 105
Mid-price: 100
Spread: 10
```

Example of processing an add (non-crossing) message

```
$ cat input/message_ex_add.txt
A c S 97 36
```

```
$ Rscript solution.R input/book_1.csv input/message_ex_add.txt
$ask
  oid price size
2   a   105  100
1   c    97   36

$bid
  oid price size
1   b    95  100

Total volume: 100 136
Best prices: 95 97
Mid-price: 96
Spread: 2
```

Example of processing a crossing add message

```
$ cat input/message_ex_cross.txt
A c B 106 101
```

```
$ Rscript solution.R input/book_1.csv input/message_ex_cross.txt
$ask
[1] oid   price size
<0 rows> (or 0-length row.names)

$bid
  oid price size
1   c   106    1
2   b    95  100

Total volume: 101 0
Best prices: 106 NA
Mid-price: NA
Spread: NA
```

Sample output

We provide sample output for 9 cases, namely all combinations of the following 3 initial books and 3 message files.

The 3 initial books are found in the `input` subdirectory and are called:

- book_1.csv
- book_2.csv
- book_3.csv

The 3 message files are also found in the `input` subdirectory and are called:

- messages_a.txt [add messages only, i.e., requires `book.add` but not `book.reduce`]
- messages_ar.txt [add and reduce messages, but no add crosses the spread]
- messages_arc.txt [add and reduce, with some adds that cross the spread]

The 9 output files can be found in the `output` subdirectory of the `comp226_a1` directory.

```
output
├── book_1-message_a.out
├── book_1-message_ar.out
├── book_1-message_arc.out
├── book_2-message_a.out
├── book_2-message_ar.out
├── book_2-message_arc.out
├── book_3-message_a.out
├── book_3-message_ar.out
└── book_3-message_arc.out

0 directories, 9 files
```

Hints for order book stats

For `book.spread` and `book.midprice` a nice implementation would use `book.best_prices`, which you should then implement first.

Hints for `book.add` and `book.reduce`

A possible way to implement `book.add` and `book.reduce` that makes use of the different example message files is the following:

- First, do a partial implementation of `book.add`, namely implement add messages that do not cross. Check your implementation with `message_a.txt`.
- Next, implement `book.reduce` fully. Check your combined (partial) implementation of `book.add` and `book.reduce` with `message_ar.txt`.
- Finally, complete the implementation of `book.add` to deal with crosses. Check your implementation with `message_arc.txt`

Hint on `book.sort`

In `comp226_a1_v3` there is a `book.sort` method, with sort code as follows:

```
book.sort <- function(book, sort_bid=T, sort_ask=T) {
    if (sort_ask && nrow(book$ask) >= 1) {
        book$ask <- book$ask[order(book$ask$price,
                                   nchar(book$ask$oid),
                                   book$ask$oid,
                                   decreasing=F),]
        row.names(book$ask) <- 1:nrow(book$ask)
    }

    if (sort_bid && nrow(book$bid) >= 1) {
        book$bid <- book$bid[order(-book$bid$price,
                                   nchar(book$bid$oid),
                                    book$bid$oid,
                                    decreasing=F),]
        row.names(book$bid) <- 1:nrow(book$bid)
    }
    book
}
```

This method will ensure that limit orders are sorted first by price and second by time of arrival (so that for two orders at the same price, the older one is nearer the top of the book).

You are welcome (and encouraged) to use `book.sort` in your own implementations. In particualar, by using it you can avoid having to find exactly where to place an order in the book.

Hint on using logging in `book.reconstruct`

In `comp226_a1_v3` a logging option has been added to `book.reconstruct`:

```
book.reconstruct <- function(data, init=NULL, log=F) {

    if (nrow(data) == 0) return(book)
    if (is.null(init)) init <- book.init()

    book <- Reduce(
         function(b, i) {
             new_book <- book.handle(b, data[i,])
             if (log) {
                 cat("Step", i, "\n\n")
                 book.summarise(new_book, with_stats=F)
                 cat("====================\n\n")
             }
             new_book
         },
         1:nrow(data), init,
    )
    book.sort(book)
}
```

You can turn on logging by changing `log=F` to `log=T`. Then `book.summarise` will be used to give output after each message is processed by `book.reconstruct`.

Submission

Remember to submit a single "MWS-username.R" file, where MWS-username should be replaced with your MWS username.