1. **[5 marks]**

```
> library(xml2)
> # read_html() NOT read_xml()
> chart <- read_html("2017-07-29.html")
> # xml_find_first() AND xml_text()
> xml_text(xml_find_first(chart, "//h2"))

[1] "Despacito"

> # xml_find_first() and xml_text() AND trim=TRUE
> xml_text(xml_find_first(chart, "//a"), trim=TRUE)

[1] "Luis Fonsi & Daddy Yankee Featuring Justin Bieber"

> # xml_find_first() and xml_text() AND XPath
> xml_text(xml_find_first(chart, "//div[@class='chart-row__rank']/span"))

[1] "1"

> # xml_find_all and XPath
> xml_text(xml_find_all(chart, "//div[@class='chart-row__last-week']/span"))

[1] "Last Week" "1"
```

2. **[5 marks]**

The **jsonlite** package provides functions for reading data in a JSON format into R data structures. JSON data tends to be hierarchical (nested), but often with a consistent repetition. For example, the following JSON structure is an array with two records, each consisting of components 'a' and 'b', and each 'b' component has sub-components 'c' and 'd'.

```
[
  {
    "a": 1,
    "b": {
           "c": 3,
           "d": 4
         }
  },
  {
    "a": 5,
    "b": {
           "c": 7,
           "d": 8
         }
  }
]
```

This structure translates to a data frame in R with two rows ...

```
> df
```

```
  a b.c b.d
1 1   3   4
2 5   7   8
```

... but because of the nesting, this is actually a nested data frame. It is a data frame with two columns ...

```
> dim(df)

[1] 2 2

> names(df)

[1] "a" "b"
```

... where the second column is itself a data frame (with 2 rows and 2 columns), corresponding to the nested 'b' component in the JSON.

```
> df$b

  c d
1 3 4
2 7 8
```

The `flatten()` function from **jsonlite** converts this sort of nested data frame into a flat data frame with two rows and three columns, which is easier to work with in R.

```
> df2 <- flatten(df)
> df2

  a b.c b.d
1 1   3   4
2 5   7   8

> dim(df2)

[1] 2 3

> names(df2)

[1] "a"   "b.c" "b.d"
```

3. **[10 marks]**

First, we make a new directory (folder) called `exam`.

```
pmur002@stats769prd01:~/$ mkdir exam
```

Next, we "change directory" into that new directory.

```
pmur002@stats769prd01:~/$ cd exam
```

Now we are listing all files within a directory elsewhere on the machine, with `-1` ensuring that the listing output has one line per file. The result is "piped" to the `wc` program, which counts the number of lines (thanks to the `-l` option). The result is a count of the number of files in the directory (98,973).

```
pmur002@stats769prd01:~/exam$ ls -1 /course/AT/BUSDATA/ | wc -l
98973
```

We start again with a file listing, of the same directory, but this time the `-l` option means that we get a "long" listing for each file, including information on permissions and, crucially, file size. The listing result is piped to an `awk` program that prints the fifth value from each line of the listing, which corresponds to the file size. The output from `awk` is redirected to a file called `sizes.txt`.

```
pmur002@stats769prd01:~/exam$ ls -l /course/AT/BUSDATA/ | awk '{ print($5) }' > sizes.txt
```

We examine the first few lines of the new file `sizes.txt`. The output is the size of the first few files in the directory we listed.

```
pmur002@stats769prd01:~/exam$ head sizes.txt

343
345
345
345
436
437
438
531
438
```

Finally, we use `grep` to search for the text `,6215,` in all files that match the pattern `trip_updates_20170401*.csv` (where `*` is a wildcard) in the directory we previously listed. The `-no-filename` option means that the output from `grep` is just the matching lines from the files that we search. The result is redirected to a file called `bus-6215-2017-04-01.csv`. Due to the nature of the data set we are working with, the file pattern selects only files that correspond to the 1st of April 2017 and the `grep` only matches lines for the bus with an ID of 6215. In other words, this is a subsetting operation.

```
pmur002@stats769prd01:~/exam$ grep --no-filename ',6215,' \
> /course/AT/BUSDATA/trip_updates_20170401*.csv > bus-6215-2017-04-01.csv
```

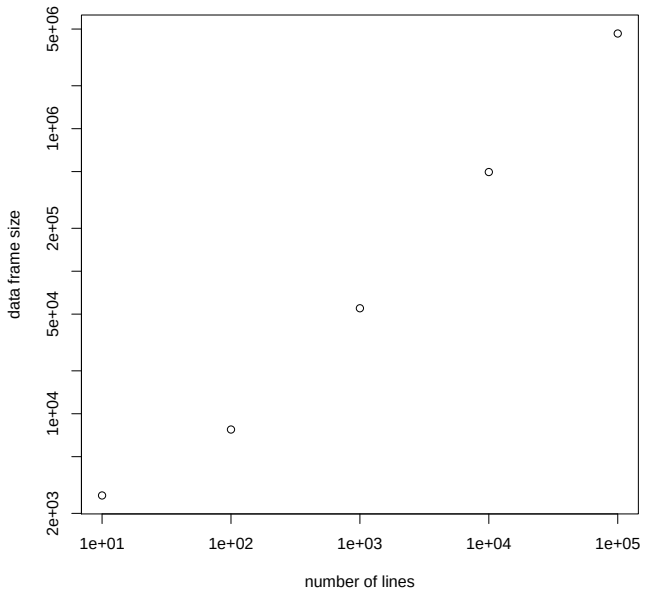
This shows the first few lines of the file that we created, showing some of the data for the bus that we have extracted.

```
pmur002@stats769prd01:~/exam$ head bus-6215-2017-04-01.csv
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,NA,252,6,7168,1490975922
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,275,NA,7,8502,1490976035
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,275,NA,7,8502,1490976035
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,275,NA,7,8502,1490976035
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,NA,293,9,8516,1490976233
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,299,NA,9,8516,1490976239
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,NA,319,10,8524,1490976349
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,388,NA,10,8524,1490976418
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,388,NA,10,8524,1490976418
8300033770-20170322104732_v52.21,30002-20170322104732_v52.21,6215,NA,403,11,8532,1490976523
```

4. **[10 marks]**

```
numLines <- 10^(1:5)
samples <- lapply(numLines,
                  function(i) {
                      read.csv("/course/AT/alldata.csv",
                               nrows=i, stringsAsFactors=FALSE)
                  })

plot(numLines, sapply(samples, object.size), log="xy",
     xlab="number of lines", ylab="data frame size")
```



The code is calling `read.csv()` on different sized subsets of the file `alldata.csv`. It then plots the size of the R object that is generated by the calls to `read.csv()`.

If we can assume that the file has the same structure and content as first 100,000 lines, then this should allow us to estimate the size of the data frame for the complete file. The plot is not exactly a straight line, but extrapolating should still give us a reasonable estimate. We might get a safer estimate by selecting a random sample of rows rather than the first n rows.

Another approach would be to look at a subset of the file and determine the class of the columns of the data frame that will be created. We could then multiply the number of rows in the file by the number of columns by the number of bytes per value in each column. For example, if a column is numeric, each value will take up 8 bytes, an integer column would be 4 bytes per value, and a character column would be 8 bytes per value. There will also be some overhead, especially if the values in a character column are all different, in which case we would need to allow for approximately one byte per character per unique character value as well.

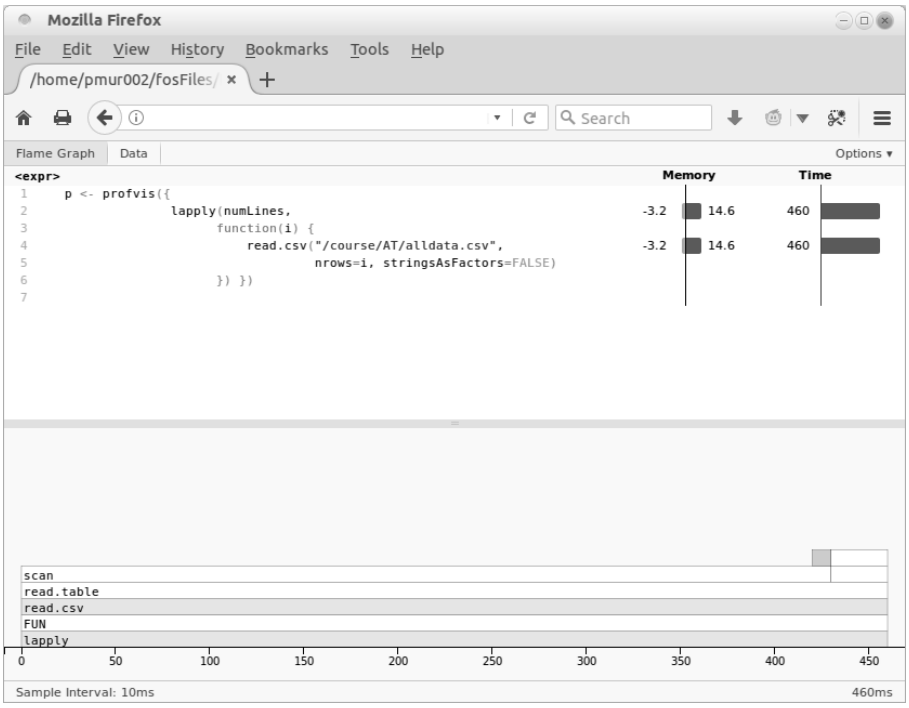
5. **[10 marks]**

```
user.self user.self user.self user.self user.self
    0.001     0.001     0.005     0.036     0.417
```

The timing results show that the time taken to read a subset of the data is increasing, approximately linearly; the number of lines in the subset is increasing by a factor of 10 each time, as is the time taken.

The second time is not ten times the first time because both of those timings are close to the resolution of timings (they are both almost zero as far as the timing system can tell).

The timings are "user" time, which means the time spent by R on the calculation. This may be different from "elapsed" time, which is the wall clock time, which is typically longer than "user" time because the computer may spend time on other things than running R's calculation.



The profiling output shows that, out of the total time spent running the R code, all of the time was spent in calls to `read.csv()`. Furthermore, all of the time spent by `read.csv()` was actually spent in a call to the `scan()` function, which is called by `read.table()`, which is called by `read.csv()`.

The best place to focus on in order to speed up the code is on the call to `read.csv()`. We could get a minor speed up by adding a `colClasses` argument to the `read.csv()` call, to specify the class for each column of data, rather than forcing `read.csv()` to spend time figuring that out for us. We could make the code run a lot faster by using a different function than `read.csv()`, e.g., `fread()` from the **data.table** package.

6. **[10 marks]**

Code for a parallel version of the code in Question 4, using `mclapply()` from the **parallel** package:

```
> library(parallel)
> samples <- mclapply(numLines,
+                     function(i) {
+                         read.csv("/course/AT/alldata.csv",
+                                  nrows=i, stringsAsFactors=FALSE)
+                     },
+                     mc.cores=5)
```

Code using `clusterApply()` from the **parallel** package:

```
> cl <- makeCluster(5)
> samples <- clusterApply(cl,
+                         numLines,
+                         function(i) {
+                             read.csv("/course/AT/alldata.csv",
+                                      nrows=i, stringsAsFactors=FALSE)
+                         })
```

The `makeCluster()` function allows us to run slave R sessions on multiple machines, but because this task involves reading files from disk, that approach would require more set up (e.g., to get files on each machine). Also, there are only 5 tasks, so there is no need to go outside a single machine, which is likely to have at least 4 cores. Also, this task is fairly quick so the overhead of `makeCluster()` starting multiple R sessions might hurt more than it helps.

The tasks are different sizes, so load balancing might help (sending tasks one at a time to slaves, rather than sending blocks of tasks to each slave), as long as we schedule the tasks from largest to smallest. However, the total time taken is very small so there is not much motivation to try it.