1. **[5 marks]**

```
> library(jsonlite)
> trips <- fromJSON("AT.json")
> trips

    id stop_time_update.stop_sequence stop_time_update.stop_id  timestamp
1 2928                            20                     7812 1474316682
2 2929                            60                     6569 1474316645

> dim(trips)

[1] 2 3

> names(trips)

[1] "vehicle"          "stop_time_update" "timestamp"
```

The `trips` object is a nested data frame, with 3 columns. Column 2 of the data frame is itself a data frame with 2 columns.

We could extract the `stop_id` information by flattening the `trips` data frame ...

```
> flatten(trips)[["stop_time_update.stop_id"]]

[1] "7812" "6569"
```

... or by extracting the second column of the second column of `trips` ...

```
> trips[["stop_time_update"]][["stop_id"]]

[1] "7812" "6569"
```

2. **[5 marks]**

Write down the result of the following R code:

```
> library(xml2)
> ird <- read_xml("IRD.xml")
> xml_text(xml_find_all(ird, "//td[@align = 'right']"))

[1] "$142.03"   "$359.77"   "$532.77"   "$1,308.78" "$431.14"   "$489.60"
[7] "$221.08"
```

Write R code to extract the first column of values from the table. Your code should produce the following result:

```
> xml_text(xml_find_all(ird, "//tr/td[1]"))

[1] "18 NCO Club"                   "1977 Masters Association"
[3] "1979 Reunion"                  "1993 Summer Camp Account"
[5] "1St Wainuiomata Venterer Unit" "44 South Travel"
[7] "81 Masters Association"
```

3. **[5 marks]**

```
# login to remote linux machine
# (using same username as current linux machine)
pmur002@sc-stat-346130:~$ ssh stats769prd01.its.auckland.ac.nz

# make a new directory called "exam"
pmur002@stats769prd01:~$ mkdir exam

# change into the new directory
pmur002@stats769prd01:~$ cd exam

# copy files from the directory /course/data/Ass2/ with names
# that start with exreg-10000. (and have any suffix)
pmur002@stats769prd01:~/exam$ cp /course/data/Ass2/exreg-10000.* .

# show detailed information about files in the current directory
# two files are almost 8MB each and one file is much smaller
# I own all files and have read/write permissions on them
# noone else can read or write the files
pmur002@stats769prd01:~/exam$ ls -lh
total 16M
-rw-rw---- 1 pmur002 pmur002 7.8M Sep 20 12:12 exreg-10000.bin
-rw-rw---- 1 pmur002 pmur002  473 Sep 20 12:12 exreg-10000.desc
-rw-rw---- 1 pmur002 pmur002 7.7M Sep 20 12:12 exreg-10000.txt

# count the number of lines (10000), words (1010000), and
# characters (7979235) in the file exreg-10000.txt
pmur002@stats769prd01:~/exam$ wc exreg-10000.txt
  10000 1010000 7979235 exreg-10000.txt

# run awk with the program 'NR < 1000' on the file exreg-10000.txt
# and redirect the result to the file exreg-sub.txt
# the result is a new file, exreg-sub.txt, which contains the first
# 999 lines from exreg-10000.txt
pmur002@stats769prd01:~/exam$ awk 'NR < 1000' exreg-10000.txt > exreg-sub.txt
```

4. **[5 marks]**

This code creates a million numeric values, which, at 8 bytes each, take up approximately 8MB of memory. This is reflected in the increase in "Vcells" for BOTH "used" and "max used" of around 8MB.

```
> gc(reset=TRUE)

         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 327319 17.5     592000 31.7   327319 17.5
Vcells 535274  4.1    1023718  7.9   535274  4.1

> x <- rnorm(1000000)
> gc()

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells  327342 17.5     592000 31.7   329099 17.6
Vcells 1535322 11.8    2613372 20.0  1537276 11.8
```

This code also creates a million numeric values, but does so within a function call. This means that the 8MB is only used within the function call, then it is released. The result is that "Vcells" "used" remains the same, but the "max used" rises by around 8MB.

The `NULL` is important because that provides the return value from the function. That return value is automatically stored by R in the variable `.Last.value`. Without the `NULL`, the return value of the function would be the million numeric values, and that would increase "used" by 8MB as well.

```
> f <- function() {
+     x <- rnorm(1000000)
+     NULL
+ }
> f()

NULL

> gc()

          used (Mb) gc trigger (Mb) max used (Mb)
Ncells  327391 17.5     592000 31.7   337299 18.1
Vcells 1535481 11.8    2613372 20.0  2543431 19.5
```

5. **[5 marks]**

```
> # ssh stats769prd01.its.auckland.ac.nz
>
> # Faster read and less memory
> f1987 <- read.csv("1987.csv",
+                   colClasses=c(rep("NULL", 3),
+                                "numeric",
+                                rep("NULL", 11),
+                                "numeric",
+                                rep("NULL", 13)))
> # Faster read and faster aggregate
> library(data.table)
> f1987 <- fread("1987.csv")
> f1987[, mean(DepDelay, na.rm=TRUE), DayOfWeek]
```

6. **[5 marks]**

```
> # Naive compute
> # This would get messy because not only is there a total of 11GB
> # of data to read in from disk, but copies will be made during the rbind()
> # (testing this on only two files, less than 1GB on disk, got RAM up to 3GB)
> filenames <- paste0(1987:2008, ".csv")
> flights <- do.call(rbind, lapply(filenames, read.csv))
> aggregate(flights["DepDelay"], list(DoW=flights$DayOfWeek), mean, na.rm=TRUE)
```

7. **[10 marks]**

```
> # Parallel compute
> library(parallel)
> sumFile <- function(x) {
+     file <- fread(x)
+     sums <- file[, sum(DepDelay, na.rm=TRUE), DayOfWeek]
+     counts <- file[, sum(!is.na(DepDelay)), DayOfWeek]
+     result <- merge(sums, counts, by="DayOfWeek")
+     colnames(result) <- c("DoW", "sum", "count")
+     result
+ }
> stats <- mclapply(filenames, sumFile, mc.cores=20)
> totals <- Reduce(function(x, y) { as.matrix(x) + as.matrix(y) }, stats)
> meanDepDelay <- cbind(totals[,1]/22, totals[,2]/totals[,3])
> # If we do NOT use pre-scheduling AND reverse the order of the files
> # (so they are roughly largest to smallest), it would run faster
> stats <- mclapply(rev(filenames), sumFile, mc.cores=20,
+                   mc.preschedule=FALSE)
```

8. **[5 marks]**

The profile results show us that the `sumFile()` function ...

- is calling `fread()` from the **data.table** package.
- spends most of its time in the `fread()` call, i.e., most of its time reading data in.
- spends most of the rest of its time in a subsetting call, subsetting a `data.table` object, which suggests that the function is using the special subsetting syntax for `data.table` objects to calculate sums and counts (and these calculations take up a tiny fraction of the time compared to reading the data in)

9. **[5 marks]**

i. *Explain why we might need to use the functions* `GET()` *or* `POST()` *from the* **httr** *package to get information from a web site, rather than just the* `download.file()` *function.*

The `GET()` and `POST()` functions allow us to send extra information besides the web site URL. This makes it possible to send authentication information for password-protected sites. It also makes it possible to send information for a web form.

ii. *Explain why we might need to use the functions* `makeCluster()` *and* `clusterApply()`*, instead of the* `mclapply()` *function.*

`mclapply()` only works on Linux, not Windows (because it relies on "forking" the R process). Also, `mclapply()` only makes use of cores on the local machine, whereas `makeCluster()` can be used to create slave R sessions on remote machines as well.