MTH 2520 R Notes 13

12 Text Manipulation and Regular Expressions

12.1 Text Manipulation

- R has several built-in functions for manipulating ***character strings*** (text):

```
tolower(),
toupper()    # Change the case of the letters in a character string
grep(),
grepl()        # Search a character vector for a specified character
               # pattern, and return the vector indices (or a logical
               # vector) indicating the matches
sub(), gsub()  # Replace the first instance (or all instances) of one
               # character pattern by another
nchar()        # Returns the number of characters in a character string
paste()        # Concatenate (combine) character strings
strsplit()     # Splits a character string into substrings according
               # to a character pattern for splitting
substr()       # Returns the substring at a specified character posi-
               # tion within a character string.  Can also be used to
               # replace the substring.
regexpr(),
gregexpr()     # Returns the character position of the first instance
               # (or all instances) of a specified character pattern
```

12.1.1 Using `tolower()` and `toupper()` to Change the Case of a Character String

- `tolower()` and `toupper()` take an argument `x`, a character string (or entire `"character"` vector), and convert capital letters to lower case or vice versa. For example:

```
> tolower(x = "The rain in Spain stays mainly in the plain")

[1] "the rain in spain stays mainly in the plain"

> toupper(x = "The rain in Spain stays mainly in the plain")

[1] "THE RAIN IN SPAIN STAYS MAINLY IN THE PLAIN"
```

12.1.2 Using `grep()` and `grepl()` to Search for a Character Pattern

- `grep()` takes arguments `pattern`, a character pattern, and `x`, a `"character"` vector, and returns the indices of the elements of `x` that contain the pattern.

  `grepl()` takes the same arguments, but returns a `"logical"` vector indicating the elements that contain the pattern.

- For example, consider the `"character"` vector:

```
> pets <- c("dog", "cat", "gerbil", "hamster", "parakeet", "goldfish", "iguana")
```

  To determine which elements of `pets` contain the pattern `"er"`, type:

```
> grep(pattern = "er", x = pets)
```

```
[1] 3 4
```

  This indicates that the 3rd and 4th elements, namely `"gerbil"` and `"hamster"`, contain `"er"`.

12.1.3 Using `sub()` and `gsub()` to Substitute One Character Pattern for Another

- `sub()` takes arguments `pattern`, a character pattern, `replacement`, another character pattern, and `x`, a character string (or entire `"character"` vector), and replaces the *first instance* of the `pattern` in `x` by the `replacement`.

  `gsub()` takes the same arguments, but replaces *all instances* of the `pattern` by the `replacement`.

- For example, consider the tongue twister in which `"pack"` was incorrectly typed (twice) instead of `"pick"`:

```
> twister <- "Peter Piper packed a peck of packled peppers"
```

```
> sub(pattern = "pack", replacement = "pick", x = twister)
```

```
[1] "Peter Piper picked a peck of packled peppers"
```

- Note that `sub()` only replaced the *first* instance of `"pack"` by `"pick"`. To replace *all* instances, use `gsub()`:

```
> gsub(pattern = "pack", replacement = "pick", x = twister)
```

```
[1] "Peter Piper picked a peck of pickled peppers"
```

12.1.4 Using `nchar()` to Count Characters

- `nchar()` counts the number of characters in a character string. For example, `"Mississippi"` has 11 characters:

```
> nchar("Mississippi")

[1] 11
```

- `nchar()` is vectorized. For example (using the `"character"` vector `pets` from above):

```
> nchar(pets)

[1] 3 3 6 7 8 8 6
```

12.1.5 Using `paste()` to Combine Character Strings

- `paste()` combines two (or more) character strings together into one character string. An optional argument, `sep`, is used to specify the character separator to use when pasting the strings together. Its default value is `" "`, which separates the terms by a blank space.
- Here's an example:

```
> paste("I", "love", "R")

[1] "I love R"
```

- If the arguments passed to `paste()` are vectors, they're combined term-by-term to give a character vector result. Vector arguments are recycled as needed. For example:

```
> paste("A", 1:6, sep = "")

[1] "A1" "A2" "A3" "A4" "A5" "A6"
```

12.1.6 Using `substr()` to Extract or Replace Character Substrings

- `substr()` takes arguments `x`, a character string (or entire `"character"` vector), and `start` and `stop`, two character positions, and returns the substring of `x` from `start` to `stop`. For example:

```
> MM <- "Mickey Mouse"

> substr(MM, start = 3, stop = 6)

[1] "ckey"
```

  This says that the four characters occupying the 3rd through 6th positions of `"Mickey Mouse"` are `"ckey"`.

- Blank spaces are considered characters, so the 7th position of `"Mickey Mouse"` is occupied by `" "`, not `"M"`.
- `substr()` is a *replacement function*, so it can also be used to replace a substring. For example:

```
> substr(MM, start = 3, stop = 6) <- "nnie"

> MM

[1] "Minnie Mouse"
```

12.1.7 Using `strsplit()` to Split Character Strings

- `strsplit()` takes arguments `x`, a character string (or entire `"character"` vector), and `split`, a character pattern, and splits the character string into substrings according to matches of `split`. It returns a list of `"character"` vectors, one for each element of `x`.
- For example, to split the tongue twister

```
> twister <- "Peter Piper picked a peck of pickled peppers"
```

into separate words (so `split=" "`), type:

```
> strsplit(twister, split = " ")

[[1]]
[1] "Peter"   "Piper"   "picked"  "a"       "peck"    "of"      "pickled"
[8] "peppers"
```

Note that in this case, because `twister` is a single character string, the return value is a list with just one element, which is a `"character"` vector.

12.1.8 Using `regexpr()` and `gregexpr()` to Search for Character Patterns

- `regexpr()` takes arguments `pattern`, a character pattern, and `text`, a character string (or entire `"character"` vector), and searches the `text` for the `pattern`, returning the starting character position of the *first match* (or -1 if there's none).

  (If `text` is a `"character"` vector, `regexpr()` returns a vector of the same length indicating the position of the first match in each element of `text`).

  `gregexpr()` takes the same arguments, but returns a vector containing the starting positions of *all matches* of the `pattern` in the `text`.

  (If `text` is a `"character"` vector, `gregexpr()` returns a list of the same length, each element of which is a vector indicating the starting positions of *all matches* of the `pattern` in the corresponding element of `text`).
- For example:

```
> twister <- "Peter Piper picked a peck of pickled peppers"

> regexpr(pattern = "pick", text = twister)

[1] 13
attr(,"match.length")
[1] 4
attr(,"useBytes")
[1] TRUE
```

This indicates that the *first match* of the pattern `"pick"` begins at the 13th character of `twister`.

(The *attribute* `match.length` gives the length of the `pattern`. The `useBytes` *attribute* indicates whether matching was done byte-by-byte, as opposed to character-by-character. See the help file.)

- Note that `regexpr()` only located the *first* instance of `"pick"`. To find *all* instances, use `gregexpr()`:

```
> gregexpr(pattern = "pick", text = twister)

[[1]]
[1] 13 30
attr(,"match.length")
[1] 4 4
attr(,"useBytes")
[1] TRUE
```

This says that `"pick"` appears twice in `twister`, once starting at the 13th character position, and a second time starting at the 30th position.

Section 12.1 Exercises

Exercise 1 Here's a famous quote from the 1948 film The Treasure of the Sierra Madre:

```
> badges <- "Badges? We ain't got no badges. We don't need no badges. I
don't have to show you any stinking badges!"
```

a) Create the character string `badges` containing the entire quote (i.e. `badges` should be a length-one vector whose only element is the whole quote). If you copy and paste into R, make sure it's all on one line so that you don't end up with a newline character, `\n`, included in the quote.

b) Use `tolower()` to convert the quote to all lower case. You should now have this:

```
> badges
```

```
[1] "badges? we ain't got no badges. we don't need no badges. i don't have to show you an
```

c) We want to "clean up" the quote a bit by removing the punctuation marks (periods, question mark, and exclamation marks).

Use `gsub()`, with `pattern="!"` and `replacement=""`, to remove the exclamation mark from `badges`.

d) Removing the question mark and periods is a bit tricky. We need to specify `pattern="\\?"` and `pattern="\\."` in the call to `gsub()` (with and `replacement=""`). Using `pattern="?"` and `pattern="."` **won't work**. (See Section 12.2 for an explanation.)

You should now have this:

```
> badges
```

```
[1] "badges we ain't got no badges we don't need no badges i don't have to show you any s
```

e) Use `strsplit()`, with `split=" "`, to split the `badges` quote into individual words. You should end up with this:

```
> badges
```

```
[[1]]
 [1] "badges"   "we"       "ain't"    "got"      "no"       "badges"
 [7] "we"       "don't"    "need"     "no"       "badges"   "i"
[13] "don't"    "have"     "to"       "show"     "you"      "any"
[19] "stinking" "badges"
```

f) Note that `strsplit()` returns a *list* with one element, which is a `"character"` vector of words from the quote. Extract the vector from the list, for example by typing:

```
> badges <- badges[[1]]
```

You should now have this `"character"` vector:

```
> badges
```

```
 [1] "badges"   "we"       "ain't"    "got"      "no"       "badges"
 [7] "we"       "don't"    "need"     "no"       "badges"   "i"
[13] "don't"    "have"     "to"       "show"     "you"      "any"
[19] "stinking" "badges"
```

g) Use `nchar()` (with your vector from part *f*) to count the number of letters (characters really) in each word.

h) Use `grep()`, with `pattern="badges"`, to find the instances of the word `"badges"` in the quote.

12.2 Regular Expressions

- ***Regular expressions*** are sequences of characters used to search for and replace character patterns in text.

  The fundamental building blocks of regular expressions are single characters, including letters and digits, that match themselves. These are called ***literal characters***.

  Literal characters can be combined with some "wildcard" characters called ***metacharacters*** that, unless preceded by a backslash, have special meaning.

- For example, the letter `"i"` is a *literal character*, but a period `"."` is a *metacharacter* that matches *any* character.

  Thus both `"pick"` and `"p.ck"` are *regular expressions*, but the first one matches only the exact pattern `"pick"`, whereas the second one matches `"pick"`, `"pack"`, `"peck"`, etc.

  Consider again the tongue twister:

```
> twister <- "Peter Piper picked a peck of pickled peppers"
```

  Recall that `gregexpr()` returns the starting character positions of all matches of a `pattern`. Thus, whereas specifying `pattern="pick"` only identifies two instances, specifying `pattern="p.ck"` identifies three (the two `"pick"`s plus the `"peck"`):

```
> gregexpr(pattern = "pick", text = twister)

[[1]]
[1] 13 30
attr(,"match.length")
[1] 4 4
attr(,"useBytes")
[1] TRUE

> gregexpr(pattern = "p.ck", text = twister)

[[1]]
[1] 13 22 30
attr(,"match.length")
[1] 4 4 4
attr(,"useBytes")
[1] TRUE
```

- Just a few of R's *metacharacters* are shown below:

```
.          # Used to match any character (except a newline character
           # "\n")
|          # Used to match either of alternative characters (e.g.
```

```
            # "(a|b)c" matches "ac" and "bc")
[ ]         # Used to match any of several characters (e.g. "[ab]"
            # matches both "a" and "b")
[^ ]        # Used to negate one or more characters (e.g. "[^ab]"
            # matches any character except "a" and "b")
\s          # Used to match (a single) white space (use "\\s")
[0-9]       # Used to match any digit (use "[0-9]")
[A-Z]       # Used to match any upper case letter (use "[A-Z]")
[a-z]       # Used to match any lower case letter (use "[a-z]")
[:alpha:]   # Used to match any alphabetic character (use "[[:alpha:]]")
[:digit:]   # Used to match any single digit (use "[[:digit:]]")
[:blank:]   # Blank characters (space and tab) (use "[[:blank:]]"),
[:space:]   # Space characters (including not only space and tab, but
            # also newline and some others) (use "[[:space:]]")
[:punct:]   # Used to match any punctuation symbol (e.g. ! " # $ % & '
            # ( ) *, + , - . / : ; < = > ? @ [  ] ^ _ `  |  ~) (use
            # "[[:punct:]]")
*           # Repetition quantifier: The preceding character or sub-
            # pattern appears 0 or more times (e.g. "(ab)*" matches any
            # single character as well as the patterns "ab",  "abab",
            # "abababab", etc.)
+           # Repetition quantifier: The preceding character or sub-
            # pattern appears 1 or more times (e.g. "(ab)+" matches the
            # patterns "ab", "abab", "abababab", etc.)
?           # Preceding character or subpattern appears 0 or 1 time
            # (e.g. "(ab)?" matches any single character as well as the
            # pattern "ab")
{n}         # Preceding character or subpattern appears exactly n times
            # (e.g. "b{3}" matches the pattern "bbb")
{m,n}       # Preceding character or subpattern appears between m and n
            # times, inclusive (e.g. "b{2,4}" matches the patterns "bb",
            # "bbb", and "bbbb").  Note that there's no space after the
            # comma.
```

For the full list, type

```
> ? regex
```

Any of these can be used in the `pattern` passed to `regexpr()`, `gregexpr()`, `sub()` and `gsub()`, and `strsplit()`.

- For example, consider the vector:

```
> pets <- c("dog", "cat", "gerbil", "hamster", "parakeet", "goldfish", "iguana")
```

To use `grep()` to search for any `pet` that includes the letter `"g"` or the letter `"h"`, type:

```
> grep(pattern = "[gh]", x = pets)
```

```
[1] 1 3 4 6 7
```

- As another example using:

```
> twister
```

```
[1] "Peter Piper picked a peck of pickled peppers"
```

to search for blank spaces, type:

```
> gregexpr(pattern = "\\s", text = twister)
```

```
[[1]]
[1]  6 12 19 21 26 29 37
attr(,"match.length")
[1] 1 1 1 1 1 1 1
attr(,"useBytes")
[1] TRUE
```

- If we want to search our text for a character that's also a *metacharacter* (e.g. if we want to search for, say, a period ".", which is a metacharacter), we need to *escape* the metacharacter status using the backslash operator:

```
\        # Used to escape a metacharacter (e.g. "\." matches a period)
```

- For example, to search the period (i.e. the symbol ".") in:

```
> date <- "Jan. 27, 2014"
```

it **doesn't work** to type

```
> regexpr(".", date)
```

Instead, we have to type:

```
> regexpr("\\.", date)
```

```
[1] 4
attr(,"match.length")
[1] 1
attr(,"useBytes")
[1] TRUE
```

The reason why we had to type *two* backslashes, i.e. `"\\."`, is that it turns out that the escape character, `"\"`, *is itself a metacharacter* that needs to be escaped.

13 Performance Enhancement: Speed and Memory

13.1 Timing the Execution of R Code

- For computationally intensive tasks, e.g. those involving very large data sets, we'll want our code to run as efficiently as possible.
- To test the speed of a chunk of R code, we use:

```
system.time()     # Returns the computation time required to execute a
                  # chunk of R code (in seconds)
```

- `system.time()` takes as its argument one or more R commands, and returns the time spent executing those commands.

  (If you're timing the execution of more than one command, enclose them in curly brackets before passing them to `system.time()`.)

- The amount of time required to run your code will depend on:
  - The computer it's being run on.
  - The number and types of other applications that are running while your R code is being executed, such as web browsers, word processors, etc.
- Here we use `system.time()` to time how long it takes to add two vectors together using a loop:

```
> x <- runif(10000)
> y <- runif(10000)
> z <- NULL                       # We could use z <- numeric(0), a length-zero
>                                 # "numeric" vector

> system.time(
+     for(i in 1:10000) {
+        z[i] <- x[i] + y[i]
+     }
+ )

   user  system elapsed
   0.01    0.00    0.02
```

  We're usually mainly interested in the *user* time.

  The three times reported by `system.time()`, all of which are in seconds, are:

  - ***User time***: The CPU[1] time spent by the *current process* (R session).

- ***System time***: The CPU time spent by the *operating system*[2] on behalf of the current process (R session) carrying out tasks that must also be carried out on behalf of other processes (applications), e.g. accepting keyboard input, printing to the screen, opening files for reading or writing, etc.
- ***Elapsed time***. This is the time you would get using a stopwatch, and includes time spent on processes (applications) unrelated to the R session (e.g. open web browsers, word processors, etc.).

[1]The ***CPU*** (central processing unit) is the computer's hardware that carries out instructions of processes (applications) and the operating system, such as performing arithmetic and logical operations and input/output tasks such as receiving messages from the keyboard, printing to the screen, and writing to a file.

[2]The ***operating system*** is the computer's software that manages processes (applications), making sure they don't interfere with each other, and performs common services for those applications such as directing input/output, e.g. from the keyboard to the screen, to or from a file, or a to printer.

The sum of the user and system times gives the total CPU time spent executing the R code. But because the system time will usually be negligibly small, **we're usually mainly interested in the *user* time**.

Section 13.1 Exercises

Exercise 2 Use `system.time()` to compare the *user* times for each of the following sets of commands (all of which do the same thing):

- ```
  > x <- NULL
  > for(i in 1:100000) x <- c(x, i)
  ```

- ```
  > x <- NULL
  > for(i in 1:100000) x[i] <- i
  ```

- ```
  > x <- NULL
  > x <- 1:100000
  ```

13.2 Writing Faster R Code

- Some things that can speed up your R code include:
  - Removing unnecessary computations from loops
  - Avoiding the use of loops by using vectorization instead
  - Using vector pre-allocation

13.2.1 Removing Unnecessary Computations from Loops and Avoiding Loops by Using Vectorization

- As an example, here are three ways to add $\sqrt{2}$ to each of 100,000 numbers.
  ```

1. Here we *unnecessarily* compute $\sqrt{2}$ 100,000 times, once during each iteration of the loop:

```
> x <- runif(100000)

> system.time(
+   for(i in 1:100000) {
+     y <- sqrt(2)         # This statement can be moved outside the loop
+     x[i] <- x[i] + y
+   }
+ )

   user  system elapsed
   0.01    0.00    0.02
```

2. We can make the code more efficient by moving the command `y <- sqrt(2)` outside the loop so that it's only execute once:

```
> x <- runif(100000)

> system.time({
+   y <- sqrt(2)            # This statement was pulled from the loop
+   for(i in 1:100000) {
+     x[i] <- x[i] + y
+   }
+ })

   user  system elapsed
   0.01    0.00    0.02
```

   (Note that the commands were enclosed in curly brackets before being passed to `system.time()`.)

3. The code can be made *much* more efficient by using *vectorization* (recall that the arithmetic operators `+`, `-`, `*`, `/`, `^`, etc. are *vectorized*):

```
> x <- runif(100000)

> system.time({
+       x <- x + sqrt(2)
+ })

   user  system elapsed
      0       0       0
```

   Loops tend to be *much* slower than vectorized computations in R.

- It turns out that function calls (like `sqrt(2)` above) are time consuming in R, in part because they involve setting up environments, sometimes several nested inside each other, to store local variables, and setting up environments is time consuming. The second loop above only calls `sqrt()` once (as opposed to 100,000 times), so it's faster than the first loop.
- The reason why vectorization is (usually) faster than looping is that vectorized computations are carried out behind the scenes in the C language, which is much faster than R. (Recall that underlying R is a suite of C functions that R invokes when it needs them).

More precisely, vectorization usually involves fewer R function calls, which as noted can be slow. For example, although it may not look like it, in the two loops above, the '+' operation actually involves making 100,000 calls to the *function* `"+"()`:

```
> "+"(2, 3)
```

```
[1] 5
```

But by using vectorization, the `"+"()` function is only called once, at which point all further computations are performed in C.

13.2.2 Vector Preallocation

- ***Vector preallocation*** refers to creating a vector *prior* to executing a loop, and then *replacing* its elements during the loop's iterations (as opposed to *recreating* the vector each iteration, lengthening it by one each time). Preallocation can speed up R code.
- As an example, here are three ways to add two vectors `x` and `y` together, storing the result in a vector `z`.

1. Here we *don't* pre-allocate space in `z`, so we have to use `c()` to *recreate* `z` each iteration of the loop, lengthening it by one each time. But recreating a vector is relatively slow:

```
> x <- runif(10000)
> y <- runif(10000)

> z <- NULL                          # We don't preallocate the elements of z

> system.time(
+   for (i in 1:10000) {
+     z <- c(z, x[i] + y[i])     # R has to recreate z entirely each iteration
+   }
+ )

   user  system elapsed
   0.17    0.00    0.17
```

2. Alternatively, we can *redimension* `z` each iteration of the loop, lengthening it by one each time. But this too is *inefficient* because it turns out that redimensioning a vector takes as much time as recreating it altogether:

```
> z <- NULL                          # We don't preallocate the elements of z

> system.time(
+   for (i in 1:10000) {
+     z[i] <- x[i] + y[i]        # R has to redimension z (change its length)
+   }                            # each iteration
+ )

   user  system elapsed
   0.02    0.00    0.02
```

3. Now we *preallocate* space for the elements of `z` *before* entering the loop, which speeds the code up considerably:

```
> z <- rep(NA, 10000)              # Now we preallocate 10,000 elements of z,
>                                  # assigning each the value NA

> system.time(
+   for (i in 1:10000) {
+     z[i] <- x[i] + y[i]          # R only has to assign to a single element
+   }                              # of z, with no redimensioning
+ )
   user  system elapsed
      0       0       0
```

Section 13.2 Exercises

Exercise 3 "Growing" a data set (of any kind) using `c()`, `cbind()`, `rbind()`, etc. in a loop is much slower than setting up a "blank" data set (*pre-allocation*) and filling it in.

Below are three ways of creating the same matrix. The first two use *pre-allocation*, and avoid "growing" the matrix, but the third "grows" the matrix in the loop using `rbind()`.

Guess which one will be slowest (and which will be fastest), then check your answer using `system.time()`:

- ```
  > n <- 10000
  > i <- 1
  > my.mat <- matrix(NA, nrow = n, ncol = 6)
  > while(i <= n) {
      z <- rnorm(40, mean = 50, sd = 15)
      my.mat[i, ] <- summary(z)          # summary() returns a six-number
      i <- i + 1                         # summary of z
    }
  ```

- ```
  > n <- 10000
  > i <- 1
  > my.mat <- matrix(NA, nrow = n, ncol = 6)
  > repeat {
      z <- rnorm(40, mean = 50, sd = 15)
      my.mat[i, ] <- summary(z)
      i <- i + 1
      if (i >= n) break
    }
  ```

- ```
  > my.mat <- NULL
  > n <- 10000
  > i <- 1
  > while(i <= n) {
      z <- rnorm(40, mean = 50, sd = 15)
      my.mat <- rbind(my.mat, summary(z))
      i <- i + 1
    }
  ```
  ```

13.3 Bytecode Compilation

- A ***bytecode compiler*** translates a so-called ***high-level*** language like R, which is closer to human spoken language and therefore easier to understand but relatively slow, into a ***low-level*** language called ***bytecode***, which is closer to the computer's ***machine instruction language*** and therefore harder to understand but much faster.
- R comes equipped with its own bytecode compiler, in the built-in package `"compiler"`, which we can use to try to speed up our code.
- We'll look at one function from the `"compiler"` package:

```
cmpfun()     # Translate (compile) a function from R code to bytecode
```

- As an example, here's a function `f()` that computes the sum of the numbers from 1 to n:

```
> f <- function(n) {
+   s <- 0
+   for(i in 1:n) {
+     s <- s + i
+   }
+   s
+ }
```

Here's how long it takes to execute:

```
> system.time(f(n = 1000000))

   user  system elapsed
   0.05    0.00    0.04
```

To translate (compile) it to bytecode, we type:

```
> library(compiler)               # The "compiler" package comes built-in with R

> cf <- cmpfun(f)
```

The bytecode-compiled version is much faster than the uncompiled version:

```
> system.time(cf(n  = 1000000))

   user  system elapsed
   0.04    0.00    0.05
```

Note, though, that even the bytecode-compiled version still isn't as fast as the built-in function `sum()`:

```
> system.time(s <- sum(as.numeric(1:1000000)))

   user  system elapsed
   0.01    0.00    0.02
```

(`sum()` requires the use of `as.double()` for very large sums.)