Contents

1 Exercise 4.1

```
X <- matrix(c(1,1,0,0, 0,0,1,1), nrow=4)
A <- matrix(c(1,0, 0,2), nrow=2)
M <- matrix(c(rep(c(1,1,0,0), 2),
              rep(c(0,0,2,2), 2)),
            nrow=4)
X; A; M
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    1    0
## [3,]    0    1
## [4,]    0    1

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    2

##      [,1] [,2] [,3] [,4]
## [1,]    1    1    0    0
## [2,]    1    1    0    0
## [3,]    0    0    2    2
## [4,]    0    0    2    2
```

```
X %*% A %*% t(X) == M
```

```
##      [,1] [,2] [,3] [,4]
## [1,] TRUE TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE TRUE
## [3,] TRUE TRUE TRUE TRUE
## [4,] TRUE TRUE TRUE TRUE
```

```
all(X %*% A %*% t(X) == M)
```

```
## [1] TRUE
```

Note that X is a numeric matrix and so can only have numeric entries. `((i==2*j)|(i==2*j-1))` is a boolean (T or F), and so is converted into a number (`1` or `0`) when inserted into the numeric matrix.

```r
# Create a zero matrix
X=matrix(rep(0, 4*2), nrow=4, ncol=2)
# Change entries selectively
for (i in c(1:4)) {
  for (j in c(1:2)) {
    X[i,j] = ((i==2*j)|(i==2*j-1))
  }
}
A = diag(c(1:2))
M = X %*% A %*% t(X)

X; A; M
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    1    0
## [3,]    0    1
## [4,]    0    1

##      [,1] [,2]
## [1,]    1    0
## [2,]    0    2

##      [,1] [,2] [,3] [,4]
## [1,]    1    1    0    0
## [2,]    1    1    0    0
## [3,]    0    0    2    2
## [4,]    0    0    2    2
```

```r
X = matrix(0, nrow=8, ncol=4)
for (i in c(1:8)) {
  for (j in c(1:4)) {
    X[i,j] = ((i==2*j)|(i==2*j-1))
  }
}
A = diag(1:4)
N = X %*% A %*% t(X)

N
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    1    0    0    0    0    0    0
## [2,]    1    1    0    0    0    0    0    0
## [3,]    0    0    2    2    0    0    0    0
## [4,]    0    0    2    2    0    0    0    0
## [5,]    0    0    0    0    3    3    0    0
## [6,]    0    0    0    0    3    3    0    0
## [7,]    0    0    0    0    0    0    4    4
## [8,]    0    0    0    0    0    0    4    4
```

2 Exercise 4.2

```r
painlevel <- function(score) {
  if (score < 5) {
    if (score < 2) {
      result <- "no pain"
    } else {
      result <- "minor pain"
    }
  } else if (score < 8) {
    result <- "moderate pain"
  } else {
    result <- "severe pain"
  }
  return(result)
}
painlevel(2)
```

```
## [1] "minor pain"
```

```r
painlevel(7)
```

```
## [1] "moderate pain"
```

```r
painlevel(9)
```

```
## [1] "severe pain"
```

`painlevel(2)` will give an output of “minor pain”, `painlevel(7)` will give an output of “moderate pain”, `painlevel(9)` will give an output of “severe pain”.

Again, nested if statements are terrible. Reorganising the code makes this much more clear:

```r
painlevel <- function(score) {
  if (score < 2) {
    return ("no pain")
  } else if (score < 5) {
    return ("minor pain")
  } else if (score < 8) {
    return ("moderate pain")
  } else {
    return ("severe pain")
  }
}
```

Using `painlevel()`, one option is to use a for loop:

```r
painvector <- function(score) {
  result <- NULL
  for (i in 1:length(score)) {
    result[i] = painlevel(score[i])
  }
  result
}
painvector(c(2,7,9))
```

```
## [1] "minor pain"    "moderate pain" "severe pain"
```

Another approach, instead of dealing with values one by one, is to use functions that handle vectors. In other words, we will *vectorise* this function.

There are several options, the most basic of which uses the `ifelse()` function.

A simple demonstration of the `ifelse()` function should suffice: given a boolean vector `test`, each value in `test` will evaluate to either a value in yes (TRUE values) or no (FALSE values).

```r
# ifelse(test, yes, no)
ifelse(c(T, F, T), c(1, 2, 3), c(4, 5, 6))
```

```
## [1] 1 5 3
```

In this case, with multiple values, only a nested `ifelse()` will work.

```r
painlevel <- function(score) {
  results <- ifelse(score < 2, "no pain",
                    ifelse(score < 5, "minor pain",
                           ifelse(score < 8, "moderate pain", "severe
                            ↪  pain")))
  results
}
painlevel(c(2,7,9))
```

```
## [1] "minor pain"    "moderate pain" "severe pain"
```

Another function to solve this problem is the `cut()` function. Again, to understand how to use these functions, look up the documentation and some examples.

```r
painlevel <- function(score) {
  cut(score, c(0, 2, 5, 8, 10),
      c("none", "minor", "moderate", "severe"),
      include.lowest=T, right=F)
}

painlevel(c(2,7,9))
```

```
## [1] minor    moderate severe
## Levels: none minor moderate severe
```

3 Exercise 4.3

```r
n = 27
ex4.3 <- data.frame(
  Subject = rep(1001:(1001+n-1), rep(3,n)),
  Treatment = rep(c("Placebo", "Low Dose", "High Dose"), n),
  Period = rep(1:3, n)
)
ex4.3
```

```
## # A tibble: 81 x 3
##    Subject Treatment Period
##      <int> <chr>      <int>
##  1    1001 Placebo        1
##  2    1001 Low Dose       2
##  3    1001 High Dose      3
##  4    1002 Placebo        1
##  5    1002 Low Dose       2
##  6    1002 High Dose      3
##  7    1003 Placebo        1
##  8    1003 Low Dose       2
##  9    1003 High Dose      3
## 10    1004 Placebo        1
## # i 71 more rows
```

4 Exercise 4.4

```
CA_result <- data.frame(
  Student_ID = 101:110,
  Test1 = c(78, 87, 67, 53, 89, 57, 83, 92, 74, 82),
  Test2 = c(68, 53, 47, 56, 75, 69, 48, 84, 83, 68),
  Test3 = c(73, 67, 75, 89, 46, 75, 67, 88, 80, 75),
  Test4 = c(93, 73, 67, 73, 52, 86, 86, 87, 53, 65),
  Test5 = c(67, 70, 91, 56, 75, 56, 81, 81, 54, 77)
)

str(CA_result)
```

```
## 'data.frame':    10 obs. of  6 variables:
##  $ Student_ID: int  101 102 103 104 105 106 107 108 109 110
##  $ Test1     : num  78 87 67 53 89 57 83 92 74 82
##  $ Test2     : num  68 53 47 56 75 69 48 84 83 68
##  $ Test3     : num  73 67 75 89 46 75 67 88 80 75
##  $ Test4     : num  93 73 67 73 52 86 86 87 53 65
##  $ Test5     : num  67 70 91 56 75 56 81 81 54 77
```

The `apply()` function takes in an *iterable object*, a *margin* or *axis*, and a *function* to apply to each entry in the margin.

For example, we have a data frame, which is 2-dimensional. Recall the indexing notation: using a comma within the indexing results in indexing over rows and columns, i.e. the 1st and 2nd dimensions of the dataframe respectively.

Specifying `MARGIN = 1` will iterate over the first dimension (the rows). In this example, it takes the mean across each row.

`MARGIN = 2` will iterate over the second dimension (the columns).

```
CA_result[,2:6]
```

```
## # A tibble: 10 x 5
##    Test1 Test2 Test3 Test4 Test5
##    <dbl> <dbl> <dbl> <dbl> <dbl>
##  1    78    68    73    93    67
##  2    87    53    67    73    70
##  3    67    47    75    67    91
##  4    53    56    89    73    56
##  5    89    75    46    52    75
##  6    57    69    75    86    56
##   7     83     48     67     86     81
##   8     92     84     88     87     81
##   9     74     83     80     53     54
##  10     82     68     75     65     77
```

```
apply(CA_result[,2:6], 1, mean)
```

```
##  [1] 75.8 70.0 69.4 65.4 67.4 68.6 73.0 86.4 68.8 73.4
```

You may also have tried to apply the mean function over the row directly, but received errors:

```
CA_result[1, 2:6]
```

```
## # A tibble: 1 x 5
##   Test1 Test2 Test3 Test4 Test5
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    78    68    73    93    67
```

```
mean(CA_result[1, 2:6])
```

```
## Warning in mean.default(CA_result[1, 2:6]): argument is not numeric or logica
## returning NA
```

```
## [1] NA
```

The reason is likely because `mean()` can only act on numeric or logical vectors, but the slice across 1 row is represented as a data frame, not a vector.

To make it work, we use `as.numeric()` to convert the data frame into a vector of numerics.

```
class(CA_result[1, 2:6])
```

```
## [1] "data.frame"
```

```
as.numeric(CA_result[1, 2:6])
```

```
## [1] 78 68 73 93 67
```

```
mean(as.numeric(CA_result[1, 2:6]))
```

```
## [1] 75.8
```

```
best3_mean <- function(scores) {
  mean(sort(scores, decreasing=T)[1:3])
}

apply(CA_result[,2:6], 1, best3_mean)
```

```
##  [1] 81.33333 76.66667 77.66667 72.66667 79.66667 76.66667 83.33333 89.00000
##  [9] 79.00000 78.00000
```

5 Exercise 4.5

Making use of the `prod()` function multiplies all entries in a vector, similar to the `sum()` function adding all entries.

Modulo ensures the input is an integer (since it is remainder 0 or 1).

```
f <- function(n) {
  if (n %% 2 == 0 & n > 0) {
    x <- seq(2, n, 2)
    return(prod(x))
  } else if (n %% 2 == 1 & n > 0) {
    x <- seq(1, n, 2)
    return(prod(x))
  }
}

f(6); f(7); f(-1); f(1.5)
```

```
## [1] 48
```

```
## [1] 105
```

There is some overlapping code in both the odd and even cases. This can actually be shortened down more, with a change in `seq()` to count down towards 1 or 2 instead of up:

```
f <- function(n) {
  if (floor(n) == n & n > 0) {
    x <- seq(n, 1, -2)
    return(prod(x))
  }
}

f(6); f(7); f(-1); f(1.5)
```

```
## [1] 48
```

```
## [1] 105
```

With a `while` loop:

```r
f <- function(n) {
  if (floor(n) == n & n > 0) {
    result = 1
    while (n > 0) {
      result = result * n
      n = n - 2
    }
    return(result)
  }
}
f(6); f(7); f(-1); f(1.5)
```

```
## [1] 48
```

```
## [1] 105
```

With a `for` loop:

```r
f <- function(n) {
  if (floor(n) == n & n > 0) {
    result = 1
    for (i in seq(n, 1, -2)) result = result * i
    return(result)
  }
}
f(6); f(7); f(-1); f(1.5)
```

```
## [1] 48
```

```
## [1] 105
```