# 2

# Univariate data

**2.1** For example:

```
p <- c(2, 3, 5, 7, 11, 13, 17, 19)
```

**2.2** The `diff` function returns the distance between fill-ups, so `mean(diff(gas))` is your average mileage per fill-up, and `mean(gas)` is the uninteresting average of the recorded mileage.

**2.3** The data may be entered in using `c` then manipulated in a natural way.

```
x <- c(2, 5, 4, 10, 8)
x^2

## [1]   4  25  16 100  64

x - 6

## [1] -4 -1 -2  4  2

(x - 9)^2

## [1] 49 16 25  1  1
```

**2.4** These can be done with

```
rep(”a”, 10)

##  [1] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a"

seq(1, 99, by=2)

##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
## [21] 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79
## [41] 81 83 85 87 89 91 93 95 97 99

rep(1:3, rep(3,3))

## [1] 1 1 1 2 2 2 3 3 3

rep(1:3, 3:1)

## [1] 1 1 1 2 2 3

c(1:5, 4:1)

## [1] 1 2 3 4 5 4 3 2 1
```

**2.5** These can be done with the following commands:

```
primes_under_20 <- c(1, 2, 3, 5, 8, 13, 21, 34)
ns <- 1:10
recips <- 1/ns
cubes <- (1:6)^3
years <- 1964:2014
subway <- c(14, 18, 23, 28, 34, 42, 50, 59, 66, 72, 79, 86, 96, 103, 110)
by25 <- seq(0,1000, by=25)
```

**2.6** We have:

```
sum(abs(rivers - mean(rivers))) / length(rivers)
```

```
## [1] 313.5508
```

To elaborate, `rivers - mean(rivers)` centers the values and is a data vector. Calling `abs` makes all the values non-negative, and `sum` reduces the result to a single number, which is then divided by the length.

**2.7** The unary minus is evaluated before the colon:

```
-1:3                                    # like (-1):3
```

```
## [1] -1  0  1  2  3
```

However, the colon is evaluated before multiplication in the latter:

```
1:2*3                                   # not like 1:(2*3)
```

```
## [1] 3 6
```

**2.8** If we know the cities starting with a "J" then this is just an exercise in indexing by the names attribute, as with:

```
precip["Juneau"]
```

```
## Juneau
##   54.7
```

Getting the cities with the names beginning with "J" can be done by sorting and inspecting, say with `sort(names(precip))`. This gives:

```
j_cities <- c("Jackson", "Jacksonville", "Juneau")
precip[j_cities]
```

```
##      Jackson Jacksonville       Juneau
##         49.2         54.5         54.7
```

The inspection of the names by scanning can be tedious for large data sets. The `grepl` function can be useful here, but requires the specification of a regular expression to indicate words that start with "J". As a teaser, here is how this could be done:

```
precip[grepl("^J", names(precip))]
```

```
##       Juneau Jacksonville      Jackson
##         54.7         54.5         49.2
```

Regular expressions are described in the help page ?regexp.

**2.9** There are many ways to do this, the following uses `paste`:

```
paste("Trial", 1:10)
```

```
##  [1] "Trial 1"  "Trial 2"  "Trial 3"  "Trial 4"  "Trial 5"
##  [6] "Trial 6"  "Trial 7"  "Trial 8"  "Trial 9"  "Trial 10"
```

**2.10** This answer will very depending on the underlying system. One answer is:

```
paste(dname, fname, sep=.Platform$file.sep)
```

```
## [1] "/Library/Frameworks/R.framework/Versions/3.2/Resources/library/UsingR/DESCRIPTION"
```

**2.11** The number of levels and number of cases are returned by:

```
require(MASS)
man <- Cars93$Manufacturer
length(man)                             # number of cases
```

```
## [1] 93
```

```
length(levels(man))                     # number of levels
```

```
## [1] 32
```

**2.12** Looking at the levels, we see that one is `rotary`, which is clearly not numeric. As for the 5-cylinder cars, we can get them as follows:

```
cyl <- Cars93$Cylinders
levels(cyl)                              # "rotary"

## [1] "3"      "4"      "5"      "6"      "8"      "rotary"

which(cyl == "5")                        # just 5 is also okay

## [1] 89 93

Cars93$Manufacturer[ which(cyl == 5) ]  # which companies

## [1] Volkswagen Volvo
## 32 Levels: Acura Audi BMW Buick Cadillac Chevrolet ... Volvo
```

**2.13** The `factor` function allows this to be done by specifying the `labels` argument:

```
mtcars$am <- factor(mtcars$am, labels=c("automatic", "manual"))
```

This produces a modified, local copy of `mtcars`. The ordering of the labels should match the following: `sort(unique(as.character(mtcars$am)))`.

**2.14** The answer is no:

```
require(HistData)
any(Arbuthnot$Female > Arbuthnot$Male)

## [1] FALSE
```

Read the help page to see how this could be construed to show the "guiding hand of a devine being."

**2.15** We have:

```
A <- c(TRUE, FALSE, TRUE, TRUE)
B <- c(TRUE, FALSE, TRUE, TRUE)
!(A & B)


## [1] FALSE  TRUE FALSE FALSE


!A | !B


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

It is not necessary to express the latter as `(!A) | (!B)`, as the unary `!` operator has higher precedence than the binary `|` operator.

**2.16** We use logical extraction for this task:

```
names(precip[precip > 50])


## [1] "Mobile"       "Juneau"       "Jacksonville" "Miami"
## [5] "New Orleans"  "San Juan"
```

**2.17** After parsing the question, it can be seen that this expression answers it:

```
m <- mean(precip)
trimmed_m <- mean(precip, trim=0.25)
any(precip > m + 1.5 * trimmed_m)


## [1] FALSE
```

A similar question is used for the algorithmic determination of "outliers" in a data set.

**2.18** The comparison of strings is done lexicographically. That is, comparisons are done character by character until a tie is broken. The comparison of characters varies due to the locale. This may be decided by ASCII codes—which yields alphabetically ordering—but need not be. See `?locale` for more detail.

**2.19** First we store the data, then we analyze it.

```
commutes <- c(17, 16, 20, 24, 22, 15, 21, 15, 17, 22)
commutes[commutes == 24] <- 18
max(commutes)

## [1] 22

min(commutes)

## [1] 15

mean(commutes)

## [1] 18.3

sum(commutes >= 20)

## [1] 4

sum(commutes < 18)/length(commutes)

## [1] 0.5
```

**2.20** We need to know that the months with 31 days are 1, 3, 5, 7, 8, 10, and 12.

```
cds <- c(79, 74, 161, 127, 133, 210, 99, 143, 249, 249, 368, 302)
longmos <- c(1, 3, 5, 7, 8, 10, 12)
long <- cds[longmos]
short <- cds[-longmos]
mean(long)

## [1] 166.5714

mean(short)

## [1] 205.6
```

**2.21** Enter in the data as follows:

```
x <- c(0.57, 0.89, 1.08, 1.12, 1.18, 1.07, 1.17, 1.38, 1.441, 1.72)
names(x) <- 1990:1999
```

Using `diff` gives

```
diff(x)
```

```
##   1991   1992   1993   1994   1995   1996   1997   1998   1999
##  0.320  0.190  0.040  0.060 -0.110  0.100  0.210  0.061  0.279
```

We can see that one year was negative:

```
which(diff(x) < 0)
```

```
## 1995
##    5
```

The jump between 1994 and 1995 was negative (there was a work stoppage that year). The percentage difference is found by dividing by `x[-10]` and multiplying by 100. (Recall that `x[-10]` is all but the tenth (10th) number of `x`). The first year's jump was the largest.

```
diff(x)/x[-10] * 100
```

```
##      1991      1992      1993      1994      1995      1996
## 56.140351 21.348315  3.703704  5.357143 -9.322034  9.345794
##      1997      1998      1999
## 17.948718  4.420290 19.361554
```

**2.22** We have:

```
mean_distance <- function(x) {
  distances <- abs(x - mean(x))
  mean(distances)
}
```

**2.23** This can be done through:

```
f <- function(x) {
  mean(x^2) - mean(x)^2
}
f(1:10)


## [1] 8.25
```

**2.24** A simple answer is just given by:

```
iseven <- function(x) x %%2 == 0
```

Then `isodd` would be:

```
isodd <- function(x) x%%2 == 1
```

The following implementation ensures integers are used, and adds names:

```
iseven <- function(x) {
  x <- as.integer(x)
  ans <- x %% 2 == 0
  setNames(ans, x)                      # add names
}
iseven(1:10)


##     1     2     3     4     5     6     7     8     9    10
## FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

Restricting a function to handle only integer inputs can be achieved by using generic functions, such as described in Appendix **??**.

**2.25** A simple implementation looks like this. One could improve it by only looking at integer factors less or equal the square-root of $x$.

```
isprime <- function(x){
  !any(x %% 2:(x-1) == 0)
}
```

Though this isn’t a terribly efficient means to generate a list of primes, it can be used to check if one number is prime.

**2.26** The package containing the data set is no longer maintained, so this problem becomes quite hard to do! Here we copy the data:

```
time <- c(169, 125, 210, 118, 117, 135, 128, 120, 122, 164, 174, 155, 120, 159,
          121, 144, 129, 136, 124, 138, 195, 141, 156, 179, 109, 112, 167, 113,
          133, 153, 141, 150, 126, 202, 165, 139, 164, 162, 171, 154, 147, 148,
          137, 144, 139, 159, 128, 181, 181, 146, 161, 157, 130, 121, 122, 135,
          150, 151, 177, 168, 180, 136, 230, 153, 275, 204, 245, 177, 187, 237,
          119, 166, 205, 167, 153, 204, 156, 303, 158, 163, 155, 80, 303, 165,
          240, 130, 190, 62, 185, 286, 167, 148, 121, 140, 124, 213, 232, 102,
          106, 177, 160, 241, 166, 145, 195, 270, 188, 253, 162, 175, 191, 495,
          194)
album <- rep(c("BBC_tapes", "Rubber_Soul", "Revolver", "Magical Mystery Tour",
               "Seargent Peper", "The White album"),
             c(31, 11, 14, 14,13,30))
beatles <- data.frame(time=time, album=album)
```

We first convert time to minutes, then compute:

```
lengths <- beatles$time / 60
c(mean=mean(lengths), median=median(lengths),
  longest=max(lengths), shortest=min(lengths))


##     mean   median  longest shortest
## 2.773009 2.616667 8.250000 1.033333
```

**2.27** We need to take a weighted mean, which we do as follows:

```
nk <- ChestSizes$count
yk <- ChestSizes$chest
n <- sum(nk)
wk <- nk/n
sum(wk * yk)


## [1] 39.83182
```

**2.28** We have

```
x <- c(80,82,88,91,91,95,95,97,98,101,106,106,109,110,111)
median(x)


## [1] 97
```

**2.29** The `LearnEDA` package is no longer available. The data for `farms` is reproduced with:

```
state <- c("Al", "Als", "Ar", "Ark", "Ca", "Col", "Conn", "De", "Fl", "Ge",
           "Ha", "Id", "Ill", "Ind", "Io", "Kan", "Ken", "Lou", "Ma", "Mary",
           "Mass", "Mi", "Minn", "Miss", "Misso", "Mon", "Neb", "Nev", "NH",
           "NJ", "NM", "NY", "NC", "ND", "Oh", "Ok", "Or", "PA", "RI", "SC",
           "SD", "Te", "Tex", "Ut", "Ver", "Vir", "Wa", "WV", "Wi", "Wy")
count <- c(48, 1, 8, 49, 89, 29, 4, 3, 45, 50, 6, 25, 79, 65, 98, 65, 91, 30,
           7, 12, 6, 53, 81, 43, 110, 28, 55, 3, 3, 10, 16, 39, 58, 31, 80,
           84, 41, 59, 1, 25, 33, 91, 227, 16, 7, 50, 40, 21, 78, 9)
farms <- data.frame(state=state, count=count)
```

The stem and leaf plot is produced by:

```
stem(farms$count)


##
##   The decimal point is 1 digit(s) to the right of the |
##
##    0 | 1133346677890266
##    2 | 155890139
##    4 | 013589003589
##    6 | 5589
##    8 | 0149118
##   10 | 0
##   12 |
##   14 |
##   16 |
##   18 |
##   20 |
##   22 | 7
```

It is hard to gauge the influence of the outlier, but otherwise, the balance point is likely in the stem labeled 4, or 4000 farms. A check shows it is 44.04.

**2.30** The value is `2.3e-4` or $2.3 \cdot 10^{-4}$:

```
2.3 * 10^(-4)


## [1] 0.00023
```

**2.31** For `firstchi` this is done as follows:

```
hist(firstchi)                 # looks like 25 or so
mean(firstchi)                 # we were pretty close ...


## [1] 23.97701
```

**2.32** This is done with

```
hist(pi2000-.1, prob=TRUE)
lines(density(pi2000))
```

This distribution is “flat,” as each digit is more or less equally likely. We subtracted 0.1 so the bins for 0 and 1 are not combined, something that is seen when making the histogram at first. Alternately, we could specify the argument `breaks=0:10-.5`.

**2.33** This is done as follows:

```
hist(normtemp$temperature)    # looks like its 98.2 -- not 98.6
mean(normtemp$temperature)


## [1] 98.24923
```

**2.34** The graphics can be produced with these commands:

```
require(MASS)
hist(DDT)
boxplot(DDT)
```

The histogram shows the data to be roughly symmetric, with one outlying value, so the mean and median should be similar and the standard deviation about three-quarters the IQR. The median and IQR can be identified on the boxplot giving estimates of 3.2 for the mean and a standard deviation a little less than 0.5. We can check with this command:

```
c(mean=mean(DDT), sd=sd(DDT))


##      mean        sd
## 3.3280000 0.4371531
```

**2.35** The `hist` function needs the data to be in a data vector, not tabulated. We pad it out using `rep`, then plot. The histogram is very symmetric.

```
x <- rep(ChestSizes$chest, ChestSizes$count)
hist(x)
```

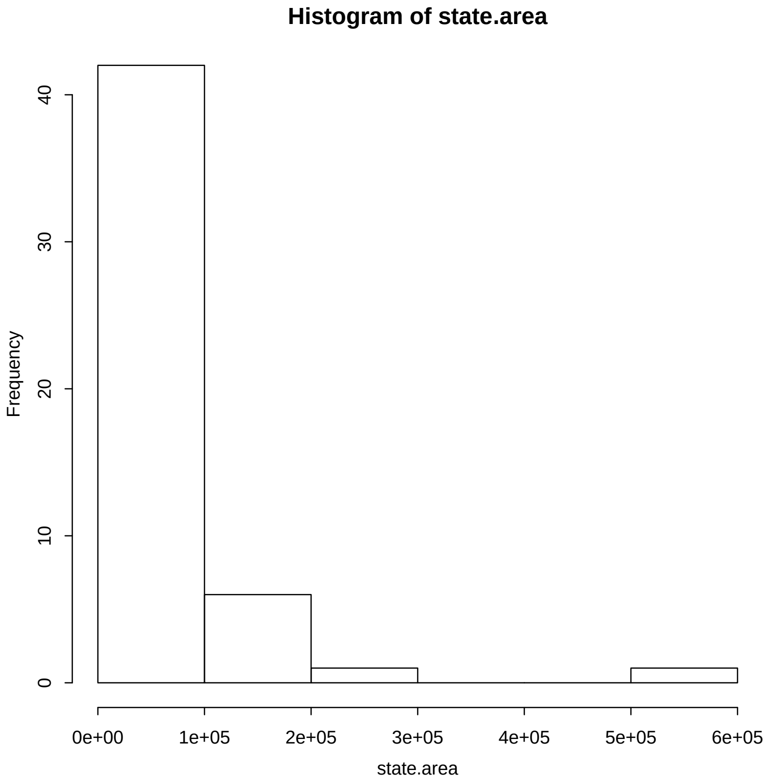
**2.36** The histogram has a rather wide range (about 3 times from smallest to largest. Some year had over 93 feet of snow fall!

**2.37** First assign names. Then you can access the entries using the respective state abbreviations.

```
names(state.area) <- state.abb
state.area[NJ]


##   NJ
## 7836


sum(state.area < state.area[NJ])/50 * 100


## [1] 8


sum(state.area < state.area[NY])/50 * 100


## [1] 40
```

To see that Alaska is the big state, we could look at this histogram then query:

```
hist(state.area)                        # 50,000 cuts of last case
state.area[state.area > 5e5]


##     AK
## 589757
```



**2.38** For a heavily skewed-right data set, such as this, the mean is significantly more than the median due to a relatively few large values. The median more accurately reflects the bulk of the data. If your intention were to make the data seem outrageously large, then a mean might be used.

**2.39** The definition of the median is incorrect. Can you think of a shape for a distribution when this is actually okay?

**2.40** The median is lower for skewed-left distributions. It makes an area look more affordable. For exclusive listings, the mean is often used to make an area seem more expensive.

**2.41** We do the usual `sum(...)/length(...)` formula:

```
sum(pi2000 <= 3)/length(pi2000) * 100


## [1] 39.5


sum(pi2000 >= 5)/length(pi2000) * 100


## [1] 50.75
```

**2.42** These values are found with:

```
sum(rivers < 500) / length(rivers)


## [1] 0.5815603


sum(rivers < mean(rivers)) / length(rivers)


## [1] 0.6666667


quantile(rivers, 0.75)


## 75%
## 680
```

**2.43** First grab the data and check the units (minutes). The top 10% is actually the 0.10 quantile in this case, as shorter times are better.

```
times <- nym.2002$time                    # easier to use
range(times)                              # looks like minutes


## [1] 147.3333 566.7833
```

```
sum(times < 3*60)/length(times) * 100   # 2.6% beat 3 hours


## [1] 2.6


quantile(times,c(.10, .25))             # 3:28 to 3:53


##     10%     25%
## 208.695 233.775


quantile(times,c(.90))                  # 5:31


##    90%
## 331.75
```

It is doubtful that the data is symmetric. It is much easier to be relatively slow in a marathon, as it requires little talent and little training—just doggedness.

**2.44** Use the functions:

```
mean(rivers)


## [1] 591.1844


median(rivers)


## [1] 425


mean(rivers, trim=.25)


## [1] 449.9155
```

Yes, the data is skewed to the right.

**2.45** We see

```
stem(islands)                               # quite skewed


##
##   The decimal point is 3 digit(s) to the right of the |
##
##    0 | 00000000000000000000000000000111111222338
##    2 | 07
##    4 | 5
##    6 | 8
##    8 | 4
##   10 | 5
##   12 |
##   14 |
##   16 | 0


c(mean=mean(islands),
  median=median(islands),
  trimmed=mean(islands,trim=0.25))


##       mean     median    trimmed
## 1252.72917   41.00000   51.08333
```

The data set is quite skewed due to the seven continents. We wouldn't expect the mean and median to agree, but note that after trimming the mean and median are similar.

**2.46** We can find the $z$-score for Barry Bonds using the name as follows:

```
(OBP[bondsba01]- mean(OBP)) / sd(OBP)


## bondsba01
##  5.990192
```

This is a decidedly "non-normal" data set, as we wouldn't expect $z$-scores larger than 3 in that case.

**2.47** Using `scale` gives us the $z$-scores.

```
z <- scale(x)[,1]                           # matrix notation
mean(z)                                     # basically 0
```

```
## [1] -1.340544e-17

sd(z)

## [1] 1
```

Alternatively, we could have found the *z*-scores directly with `(x - mean(x))/sd(x)`.

**2.48** No, as the data is skewed heavily to the right, the standard deviation is quite different:

```
c(mad=mad(exec.pay), IQR=IQR(exec.pay), sd=sd(exec.pay))

##      mad      IQR       sd
##  20.7564  27.5000 207.0435
```

**2.49** As this distribution has a long tail, we find that the mean is much more than the median.

```
amt <- npdb$amount
summary(amt)

##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##       50     8750    37500   166300   175000 25000000

sum(amt < mean(amt))/length(amt) * 100

## [1] 74.90069
```

**2.50** The value is relatively close to 1, which is the value for exponentially distributed data:

```
sd(rivers) / mean(rivers)

## [1] 0.8353922
```

**2.51** Yes, fairly close:

```
ia_times <- diff(babyboom$running.time)
sd(ia_times) / mean(ia_times)


## [1] 0.8889017
```

**2.52** The skew of `wt` is negative indicating a slight left skew. The skew of the inter-arrival times is twice as much and to the right.

```
skew(babyboom$wt)


## [1] -1.078636


ia_times <- diff(babyboom$running.time)
skew(ia_times)


## [1] 1.829281
```

Inter-arrival times are typically exponentially distributed. As such, they should have a coefficient of variation that is nearly 1.

**2.53** The histograms are all made in a similar manner to this:

```
hist(hall.fame$HR)
```

The home run distribution is skewed right, the batting average is fairly symmetric, and on-base percentage is also symmetric but may have longer tails than the batting average.

**2.54** After a log transform the data looks more symmetric. If you find the median of the transformed data, you can take its exponential to get the median of the untransformed data. Not so with the mean.

**2.55** The data is tabulated, so we first create a data vector through `rep`, then plot.

```
require(HistData)
chest <- rep(ChestSizes$chest, ChestSizes$count)
qqnorm(chest)
```

The steps are due to truncation in the measurement. Jittering can smooth this out (try `qqnorm(jitter(chest,3))`). This data hews closely to a line, and was used by Quetelet in 1846 to demonstrate "normally-distributed" data.

**2.56** The graphic is made with `qqnorm(Michelson$velocity)`. It falls fairly close to a straight line.

**2.57** The histograms can be made in a manner similar to this:

```
hist(cfb$AGE)
```

After making the graphics, we see that `AGE` is short-tailed and somewhat symmetric; `EDUC` is not symmetric; `NETWORTH` is very skewed (some can get *really* rich, some can get *pretty* poor, most close to 0 on this scale; and `log(SAVING + 1)` is symmetric except for a spike of people at 0 who have no savings (the actual data is skewed—the logarithm changes this).

**2.58** The histogram (`hist(brightness)`) shows a fairly symmetric distribution centered near 8. (Star brightness is measured on a logarithmic scale—a difference of 5 is a factor of 100 in terms of brightness. Thus, the actual brightnesses are skewed.)

**2.59** The `Price` variable is:

```
skew(Cars93$Price) > skew(Cars93$MPG.highway)


## [1] TRUE
```

(Both are skewed right.)

**2.60** This can be done as follows (using a different name from the built-in `mode` function):

```
Mode <- function(x) {
  tbl <- table(x)
  ind <- which(tbl == max(tbl))
  vals <- names(ind)
  as(vals, class(x)[1])                    # unnecessary!
}
```

Outside of the last line, this is a simple translation of the example give. The last line is not necessary. It simply generalizes the call to `as.numeric` in the example by coercing the output to the class of the input variable.

**2.61** From this command we see the answer is 1001-2000 dollars:

```
bumps <- cut(bumpers, c(0, 1000, 2000, 3000, 4000))
table(bumps)


## bumps
##     (0,1e+03] (1e+03,2e+03] (2e+03,3e+03] (3e+03,4e+03]
##             2             8             7             6
```

**2.62** The output of `summary` is a table:

```
summary(Cars93$Cylinders)


##      3      4      5      6      8 rotary
##      3     49      2     31      7      1
```

This seems a good choice—factors are used to categorize values and a table of counts is a useful summary.

**2.63** Applying the idiom to `lorem` we have:

```
chars <- unlist(strsplit(lorem, split=""))
table(chars)


## chars
##  \n       ,   ;   .   a   A   b   c   C   d   D   e   E   f   F
##  10 589  48   1  73 251   3  38 156   5 102   8 370   3  22   2
##   g   h   i   I   j   l   L   m   M   n   N   o   p   P   q   Q
##  45  17 343   8   4 220   3 142   7 211  13 174  80   6  30   2
##   r   s   S   t   u   U   v   V   x
## 183 272   7 289 289   3  49   5   3
```

Scanning we see that `e` is the most common. To avoid scanning, the function `sort` can be called on the output of `table`:

```
sort(table(chars))
```

```
## chars
##   ;   F   Q   A   E   L   U   x   j   C   V   P   M   S   D   I
##   1   2   2   3   3   3   3   3   4   5   5   6   7   7   8   8
##  \n   N   h   f   q   b   g   ,   v   .   p   d   m   c   o   r
##  10  13  17  22  30  38  45  48  49  73  80 102 142 156 174 183
##   n   l   a   s   t   u   i   e
## 211 220 251 272 289 289 343 370 589
```

**2.64** This is done with

```
require(MASS)
dotchart(table(Cars93$Cylinders))
```

```
## Warning in dotchart(table(Cars93$Cylinders)): 'x' is neither a
vector nor a matrix: using as.numeric(x)
```

The graphic shows that 4-cylinder cars were the most popular in 1993. Was this the case in 1974 (cf. `mtcars$cyl`)?

**2.68** It contains the days when nothing much happened.