8

Data Aggregation

R provides a wide array of functions to aid in aggregating data. For simple tabulation and cross-tabulation, the `table` function is available. For more complex tasks, the available functions can be broken down into two groups: those that are designed to work effectively with arrays and/or lists, like `apply`, `sweep`, `mapply`, `sapply`, and `lapply`, and those that are oriented toward data frames (like `aggregate` and `by`). There is considerable overlap between the two tools, and the output of one can be converted to the equivalent of the output from another, so often the choice of an appropriate function is a matter of personal taste.

We'll start by looking at the `table` function, and then study the other functions which can be used to aggregate data from various sources.

8.1 `table`

The arguments to the `table` function can either be individual vectors representing the levels of interest, or a list or data frame composed of such vectors. The result from `table` will always be an array of as many dimensions as the number of vectors being tabulated, with dimnames extracted from the levels of the cross-tabulated variables. By default, table will not include missing values in its output; to override this, use the `exclude=NULL` argument. When passed a single vector of values, `table` returns an object of class `table`, which can be treated as a named vector. For simple queries regarding individual levels of a tabulated variable, this may be the most convenient form of displaying and storing the values:

```
> pets = c('dog','cat','duck','chicken','duck','cat','dog')
> tt = table(pets)
```

```
> tt
pets
    cat chicken     dog    duck
      2       1       2       2
> tt['duck']
duck
   2
> tt['dog']
dog
  2
```

Alternatively, the output from `table` can be converted to a data frame using `as.data.frame`:

```
> as.data.frame(tt)
     pets Freq
1     cat    2
2 chicken    1
3     dog    2
4    duck    2
```

When multiple vectors are passed to `table`, an array of as many dimensions as there are vectors is returned. For this example, the `state.region` and `state.x77` datasets are used, creating a table that shows the number of states whose income is above and below the median income for all states, broken down by region:

```
> hiinc = state.x77[,'Income'] > median(state.x77[,'Income'])
> stateinc = table(state.region,hiinc)
> stateinc
               hiinc
state.region    FALSE TRUE
  Northeast         4    5
  South            12    4
  North Central     5    7
  West              4    9
```

This result can be converted to a data frame using `as.data.frame`:

```
> as.data.frame(stateinc)
   state.region hiinc Freq
1     Northeast FALSE    4
2         South FALSE   12
3 North Central FALSE    5
4          West FALSE    4
5     Northeast  TRUE    5
6         South  TRUE    4
7 North Central  TRUE    7
8          West  TRUE    9
```

When passed a data frame, table treats each column as a separate variable, resulting in a table that effectively counts how often each row appears in the data frame. This can be especially useful when the result of table is passed to as.data.frame, since its form will be similar to the input data frame. To illustrate, consider this small example:

```
> x = data.frame(a=c(1,2,2,1,2,2,1),b=c(1,2,2,1,1,2,1),
+                c=c(1,1,2,1,2,2,1))
> x
  a b c
1 1 1 1
2 2 2 1
3 2 2 2
4 1 1 1
5 2 1 2
6 2 2 2
7 1 1 1
> as.data.frame(table(x))
  a b c Freq
1 1 1 1    3
2 2 1 1    0
3 1 2 1    0
4 2 2 1    1
5 1 1 2    0
6 2 1 2    1
7 1 2 2    0
8 2 2 2    2
```

Since the data frame was formed from a table, all possible combinations, including those with no observations, are included.

Sometimes it is helpful to display the margins of a table, that is, the sum of each row and/or column, in order to understand differences among the levels of the variables from which the table was formed. The addmargins function accepts a table and returns a similar table, with the requested margins added. To specify which dimensions should have margins added, the margin= argument accepts a vector of dimensions; a value of 1 in this vector means a new row with the margins for the columns will be added, and a value of 2 corresponds to a new column containing row margins. The default operation to create the margins is to use the sum function. If some other function is desired, it can be specified through the FUN= argument. When a margin is added, the dimnames for the table are adjusted to include a description of the margin. As an example of the use of addmargins, consider the infert dataset, which contains information about the education and parity of experimental subjects. First, we can generate a cross-tabulation in the usual way:

```
> tt = table(infert$education,infert$parity)
```

```
> tt

          1  2  3  4  5  6
0-5yrs    3  0  0  3  0  6
6-11yrs  42 42 21 12  3  0
12+ yrs  54 39 15  3  3  2
```

To add a row of margins, we can use the following call to `addmargins`:

```
> tt1 = addmargins(tt,1)
> tt1

          1  2  3  4  5  6
0-5yrs    3  0  0  3  0  6
6-11yrs  42 42 21 12  3  0
12+ yrs  54 39 15  3  3  2
Sum      99 81 36 18  6  8
```

To add margins to both rows and columns, use a `margin=` argument of `c(1,2)`:

```
> tt12 = addmargins(tt,c(1,2))
> tt12

           1   2   3   4   5   6 Sum
0-5yrs     3   0   0   3   0   6  12
6-11yrs   42  42  21  12   3   0 120
12+ yrs   54  39  15   3   3   2 116
Sum       99  81  36  18   6   8 248
> dimnames(tt12)
[[1]]
[1] "0-5yrs"  "6-11yrs" "12+ yrs" "Sum"

[[2]]
[1] "1"   "2"   "3"   "4"   "5"   "6"   "Sum"
```

Notice that the dimnames for the table have been updated.

When it's desired to have a table of proportions instead of counts, one strategy would be to use the `sweep` function (Section 8.4) dividing each row and column by its corresponding margin. The `prop.table` function provides a convenient wrapper around this operation. `prop.table` accepts a table, and a `margin=` argument, and returns a table of proportions. With no value specified for `margin=`, the sum of all the cells in the table will be 1; with `margin=1`, each row of the resulting table will add to 1, and with `margin=2`, each column will add to 1. Continuing with the previous example, we can convert our original table to one containing proportions, having each column add to 1, as follows:

```
> prop.table(tt,2)

                1       2       3       4       5       6
  0-5yrs  0.03030 0.00000 0.00000 0.16667 0.00000 0.75000
  6-11yrs 0.42424 0.51852 0.58333 0.66667 0.50000 0.00000
  12+ yrs 0.54545 0.48148 0.41667 0.16667 0.50000 0.25000
```

For tables with more than two dimensions, it may be useful to present the table in a "flattened" form using the `ftable` function. To illustrate, consider the `UCBAdmissions` dataset, which is already a table with counts for admission to various departments based on gender. As a three-dimensional table, it would normally be displayed as a series of two-dimensional tables. Using `ftable`, the same information can be displayed in a more compact form:

```
> ftable(UCBAdmissions)
                 Dept   A   B   C   D   E   F
Admit    Gender
Admitted Male         512 353 120 138  53  22
         Female        89  17 202 131  94  24
Rejected Male         313 207 205 279 138 351
         Female        19   8 391 244 299 317
```

The `xtabs` function can produce similar results to the `table` function, but uses the formula language interface. For example, the state income by region table could be reproduced using statements like these:

```
> xtabs(~state.region + hiinc)
               hiinc
state.region    FALSE TRUE
  Northeast         4    5
  South            12    4
  North Central     5    7
  West              4    9
```

If a variable is given on the left-hand side of the tilde (`~`), it is interpreted as a vector of counts corresponding to the values of the variables on the right-hand side, making it very easy to convert already tabulated data into R's notion of a table:

```
> x = data.frame(a=c(1,2,2,1,2,2,1),b=c(1,2,2,1,1,2,1),
+                c=c(1,1,2,1,2,2,1))
> dfx = as.data.frame(table(x))
> xtabs(Freq ~ a + b + c,data=dfx)
, , c = 1

   b
a   1 2
  1 3 0
  2 0 1
```

```
, , c = 2

   b
a   1 2
  1 0 0
  2 1 2
```

8.2 Road Map for Aggregation

When confronted with an aggregation problem, there are three main considerations:

1. How are the groups that divide the data defined?
2. What is the nature of the data to be operated on?
3. What is the desired end result?

Thinking about these issues will help to point you to the most effective solution for your needs. The following paragraphs should help you make the best choice.

Groups defined as list elements. If the groups you're interested in are already organized as elements of a list, then `sapply` or `lapply` (Section 8.3) are the appropriate functions; they differ in that `lapply` always returns a list, while `sapply` may simplify its output into a vector or array if appropriate. This is a very flexible approach, since the entire data frame for each group is available. Sometimes, if other methods are inappropriate, you can first use the `split` function to create a suitable list for use with `sapply` or `lapply` (Section 8.5).

Groups defined by rows or columns of a matrix. When the goal is to operate on each column or row of a matrix, the `apply` function (Section 8.4) is the logical choice. `apply` will usually return its results as a vector or array, but will return a list if the results of operating on the rows or columns are of different dimensions.

Groups based on one or more grouping variables. A wide array of choices is available for the very common task of operating on subsets of data based on the value of a grouping variable. If the computations you desire each involve only a single vector and produce a single scalar as a result (like calculating a scalar-valued statistic for a variable or set of variables), the `aggregate` function (Section 8.5) is the most likely choice. Since aggregate always returns a data frame, it is especially useful if the desired result is to create a plot or fit a statistical model to the aggregated data.

If your computations involve a single vector, but the result is a vector (for example, a set of quantiles or a vector of different statistics), `tapply` (Section 8.5) is one available option. Unlike `aggregate`, `tapply` returns its results in a vector or array for which individual elements are easy to access,

but may produce a difficult-to-interpret display for complex problems. Another approach to the problem is provided by the `reshape` package, available through CRAN, and documented in Section 8.6. It uses a formula interface, and can produce output in a variety of forms.

When the desired result requires access to more than one variable at a time (for example, calculating a correlation matrix, or creating a scatter plot), row indices can be passed to `tapply` to extract the appropriate rows corresponding to each group. Alternatively, the `by` function can be used. Unlike `tapply`, the special list returned by `by` has a print method which will always produce an easily-readable display of the aggregation, but accessing individual elements of the returned list may be inconvenient. Naturally, for tasks like plotting, there is no clear reason to choose one approach over the other.

As mentioned previously, using `split` and `sapply`/`lapply` is a good solution if you find that other methods don't provide the flexibility you need. Finally, if nothing else seems to work, you can write a loop to iterate over the values returned by `unique` or `intersection`, and perform whatever operations you desire. If you take this route, make sure to consider the issues about memory management in loops found in Section 8.7.

8.3 Mapping a Function to a Vector or List

Although most functions in R will automatically operate on each element of a vector, the same is not true for lists. Since many R functions return lists, it's often useful to process each list element in the same way that R naturally does for vectors. To handle situations like this, R provides two functions: `lapply` and `sapply`. Each of these functions takes a list or vector as its first argument, and a function to be applied to each element as its second argument. The difference between the two functions is that `lapply` will always return its result as a list, while `sapply` will simplify its output to a vector or matrix if possible. For example, suppose we have a vector of character strings, and we want to find out how many words are in each vector. Like most functions in R, the `strsplit` function will operate on each element of a vector, returning for each element a new vector containing the individual pieces of that element:

```
> text = c('R is a free environment for statistical analysis',
+            'It compiles and runs on a variety of platforms',
+            'Visit the R home page for more information')
> result = strsplit(text,' ')
> result
[[1]]
[1] "R"           "is"          "a"
[4] "free"        "environment" "for"
[7] "statistical" "analysis"
```

```
[[2]]
[1] "It"        "compiles"  "and"
[4] "runs"      "on"        "a"
[7] "variety"   "of"        "platforms"

[[3]]
[1] "Visit"       "the"         "R"
[4] "home"        "page"        "for"
[7] "more"        "information"
```

Since each vector could potentially contain different numbers of words, `strsplit` puts its result into a list. The `length` function will not automatically operate on each list element; instead, it properly reports the number of elements in the returned list:

```
> length(result)
[1] 3
```

To find the length of the individual elements, we can use either `sapply` or `lapply`; since the length of each element will be a scalar, `sapply` would be most appropriate:

```
> nwords = sapply(result,length)
> nwords
[1] 8 9 8
```

Another important use of sapply relates to data frames. When treated as a list, each column of a data frame retains its mode and class. Suppose we're working with the built-in `ChickWeight` data frame, and we wish to learn more about the nature of each column. Simply using the `class` function on the data frame will give information about the data frame, not the individual columns:

```
> class(ChickWeight)
[1] "nfnGroupedData" "nfGroupedData"
[3] "groupedData"    "data.frame"
```

To get the same information for each variable, use `sapply`:

```
> sapply(ChickWeight,class)
$weight
[1] "numeric"

$Time
[1] "numeric"

$Chick
[1] "ordered" "factor"

$Diet
[1] "factor"
```

Notice that in this case, since the class for `Chick` was of length 2, `sapply` returned its result as a list. This will always be the case when the structure of the data would be lost if `sapply` tried to simplify it into a vector or array.

This same idea can be used to extract columns of a data frame that meet a particular condition. For example, to create a data frame containing only numeric variables, we could use

```
df[,sapply(df,class) == 'numeric']
```

`sapply` or `lapply` can be used as an alternative to loops for performing repetitive tasks. When you use these functions, they take care of the details of deciding on the appropriate form of the output, and eliminate the need to incrementally build up a vector or matrix to store the result. To illustrate, suppose that we wish to generate matrices of random numbers and determine the highest correlation coefficient between any of the variables in the matrix. The first step is to create a function that will generate a single matrix and calculate the maximum correlation coefficient:

```
maxcor = function(i,n=10,m=5){
   mat = matrix(rnorm(n*m),n,m)
   corr = cor(mat)
   diag(corr) = NA
   max(corr,na.rm=TRUE)
}
```

Since `sapply` will always pass an argument to the applied function, a dummy argument (`i`) is added to the function. Since the diagonal of a correlation matrix will always be 1, the diagonal elements of the correlation matrix were masked by assigning them values of `NA`. Suppose we want to generate 1000 100×5 matrices, and find the average value of the maximum correlation:

```
> maxcors = sapply(1:1000,maxcor,n=100)
> mean(maxcors)
[1] 0.1548143
```

Notice that additional arguments to the function being applied (like `n=100` in this case) are passed to the function by including them in the argument list after the function name or definition.

For simpler simulations of this type, the `replicate` function can be used. This function takes as its first argument the number of replications desired, and as its second argument an expression (not a function!) that calculates the desired statistic for the simulation. For example, we can generate a single t-statistic from two groups of normally distributed observations with the following expression:

```
> t.test(rnorm(10),rnorm(10))$statistic
        t
0.2946709
```

Using replicate, we can generate as many of these statistics as we want:

```
> tsim = replicate(10000,t.test(rnorm(10),rnorm(10))$statistic)
> quantile(tsim,c(0.5,0.75,0.9,0.95,0.99))
       50%        75%        90%        95%        99%
0.00882914 0.69811345 1.36578668 1.74995603 2.62827515
```

8.4 Mapping a function to a matrix or array

When your data has the added organization of an array, R provides a convenient way to operate on each dimension of the data through the `apply` function. This function requires three arguments: the array on which to perform the operation, an index telling `apply` which dimension to operate on, and the function to use. Like `sapply`, additional arguments to the function can be placed at the end of the argument list. For matrices, a second argument of 1 means "operate on the rows", and 2 means "operate on the columns".

One common use of `apply` is in conjunction with functions like `scale`, which require summary statistics calculated for each column of a matrix. Without additional arguments , the `scale` function will subtract the mean of each column and divide by the standard deviation, resulting in a matrix of z-scores. To use other statistics, appropriate vectors of values can be calculated using `apply` and provided to `scale` using the `center=` and `scale=` arguments. For example, by providing a vector of medians for centering, and a vector of mean average deviations for scaling, an alternative standardization to z-scores can be performed. Using the built-in `state.x77` dataset, we could perform such a transformation as follows:

```
> sstate = scale(state.x77,center=apply(state.x77,2,median),
+                          scale=apply(state.x77,2,mad))
```

Similar to `sapply`, `apply` will try to return its results in a vector or matrix when appropriate, making it useful in cases where several quantities need to be calculated for each row or column of a matrix. Suppose we wish to produce a matrix containing the number of nonmissing observations, the mean and the standard deviation for each column of a matrix. The first step is writing a function which will return what we want for a single column:

```
summfn = function(x)c(n=sum(!is.na(x)),mean=mean(x),sd=sd(x))
```

Now we can apply the function to a data frame with all numeric columns, or a numeric matrix like `state.x77`:

```
> x = apply(state.x77,2,sumfun)
```

```
> t(x)
             n       mean           sd
Population 50  4246.4200 4.464491e+03
Income     50  4435.8000 6.144699e+02
Illiteracy 50     1.1700 6.095331e-01
Life Exp   50    70.8786 1.342394e+00
Murder     50     7.3780 3.691540e+00
HS Grad    50    53.1080 8.076998e+00
Frost      50   104.4600 5.198085e+01
Area       50 70735.8800 8.532730e+04
```

This example illustrates another advantage of using `apply` instead of a loop, namely, that `apply` will use names that are present in the input matrix or data frame to properly label the result that it returns.

One further use of `apply` is worth mentioning. If a vector needs to be processed in non-overlapping groups, it is sometimes easiest to temporarily treat the vector as a matrix, and use `apply` to operate on the groups. For example, suppose we wish to take the sum of every three adjacent values in a vector. By first forming a three-column matrix, we can process the groups conveniently using `apply`:

```
> x = 1:12
> apply(matrix(x,ncol=3,byrow=TRUE),1,sum)
[1]  6 15 24 33
```

The `apply` function is very general, and for certain applications, there may be more efficient methods available to perform the necessary computations. For example, if the statistic to be calculated is the sum or the mean, matrix computations will be more efficient than calling `apply` with the appropriate function. In cases like this, the `rowSums`, `colSums`, `rowMeans`, or functions can be used. Each of these functions accepts a matrix (or a data frame which will be coerced to a matrix), and an optional `na.rm=` argument to specify the handling of missing values. Since these functions will accept logical values as input as well as numeric values, they can be very useful for counting operations.

For example, consider the dataset `USJudgeRatings`, which has ratings for 43 judges in twelve categories. To get the mean rating for each category, the `colMeans` function could be used as follows:

```
> mns = colMeans(USJudgeRatings)
> mns
    CONT     INTG     DMNR     DILG     CFMG
7.437209 8.020930 7.516279 7.693023 7.479070
    DECI     PREP     FAMI     ORAL     WRIT
7.565116 7.467442 7.488372 7.293023 7.383721
    PHYS     RTEN
7.934884 7.602326
```

To count the number of categories for which each judge received a score of 8 or greater, the `rowSums` function can be used by providing the appropriate logical matrix:

```
> jscore = rowSums(USJudgeRatings >= 8)
> head(jscore)
 AARONSON,L.H. ALEXANDER,J.M. ARMENTANO,A.J.
             1              8              1
   BERDON,R.I.   BRACKEN,J.J.     BURNS,E.B.
            11              0             10
```

A common situation when processing a matrix by rows or columns is that each row or column needs to be processed differently, based on the values of an auxiliary vector which already exists. In cases like this, the `sweep` function can be used. Like `apply`, the first two arguments to `sweep` are the matrix to be operated on and the index of the dimension to be used for repetitive processing. In addition, `sweep` takes a third argument representing the vector to be used when processing each column, and finally a fourth argument providing the function to be used. `sweep` operates by building matrices which can be operated on in a single call, so, unlike `apply`, only functions which can operate on arrays of values can be passed to `sweep`. All of the built-in binary operators, such as addition (`"+"`), subtraction (`"-"`), multiplication (`"*"`), and division (`"/"`) can be used, but, in general, it will be necessary to make sure an arbitrary function will work properly with `sweep`. For example, suppose we have a vector representing the maximum value found in each column of a matrix, and we wish to divide each column of the matrix by its corresponding maximum. Using the `state.x77` data frame, we could use `sweep` as follows:

```
> maxes = apply(state.x77,2,max)
> swept = sweep(state.x77,2,maxes,"/")
> head(swept)
           Population    Income Illiteracy  Life Exp    Murder
Alabama    0.17053496 0.5738717  0.7500000 0.9381793 1.0000000
Alaska     0.01721861 1.0000000  0.5357143 0.9417120 0.7483444
Arizona    0.10434947 0.7173397  0.6428571 0.9585598 0.5165563
Arkansas   0.09953769 0.5349169  0.6785714 0.9600543 0.6688742
California 1.00000000 0.8098179  0.3928571 0.9743207 0.6821192
Colorado   0.11986980 0.7733967  0.2500000 0.9790761 0.4503311
             HS Grad      Frost       Area
Alabama    0.6136701 0.10638298 0.08952178
Alaska     0.9910847 0.80851064 1.00000000
Arizona    0.8632987 0.07978723 0.20023057
Arkansas   0.5928678 0.34574468 0.09170562
California 0.9301634 0.10638298 0.27604549
Colorado   0.9494799 0.88297872 0.18319233
```

Now suppose that we wish to calculate the mean value for each variable using only those values which are greater than the median for that variable. We can calculate the medians using `apply` and then write a simple function to find the mean of the values we're interested in.

```
> meds = apply(state.x77,2,median)
> meanmed = function(var,med)mean(var[var>med])
> meanmed(state.x77[,1],meds[1])
[1] 7136.16
> meanmed(state.x77[,2],meds[2])
[1] 4917.92
```

Although the function works properly for individual columns, it returns only a single value when used in conjunction with `sweep`:

```
> sweep(state.x77,2,meds,meanmed)
[1] 15569.75
```

The source of the problem is the inequality used to subset the variable values, since it will not properly operate on the array that `sweep` produces to calculate its results. In cases like this, the `mapply` function can be used. By converting the input matrix to a data frame, each variable in the input will be processed in parallel to the vector of medians, providing the desired result:

```
> mapply(meanmed,as.data.frame(state.x77),meds)
[1]   7136.160   4917.920      1.660     71.950     10.544
[6]     59.524    146.840 112213.400
```

By default, `mapply` will always simplify its results, as in the previous case where it consolidated the results in a vector. To override this behavior, and return a list with the results of applying the supplied function, use the `SIMPLIFY=FALSE` argument.

8.5 Mapping a Function Based on Groups

To calculate scalar data summaries of one or more columns of a data frame or matrix, the `aggregate` function can be used. Although this function is limited to returning scalar values, it can operate on multiple columns of its input argument, making it a natural choice for data summaries for multiple variables. The first argument to `aggregate` is a data frame or matrix containing the variables to be summarized, the second argument is a list containing the variables to be used for grouping, and the third argument is the function to be used to summarize the data. For example, the `iris` dataset contains the values of four variables measured on a variety of samples from three species of irises. To find the means of all four variables broken down by species, `aggregate` can be called as follows:

```
> aggregate(iris[-5],iris[5],mean)
     Species Sepal.Length Sepal.Width Petal.Length Petal.Width
1     setosa        5.006       3.428        1.462       0.246
2 versicolor        5.936       2.770        4.260       1.326
3  virginica        6.588       2.974        5.552       2.026
```

Since the second argument must be a list, when a data frame is being processed it is often convenient to refer to the grouping columns using single bracket subscripts, since columns accessed this way will naturally be in the form of a list. In addition, with more than one grouping variable, specifying the columns this way will insure that the grouping variables' names will be automatically transfered to the output data frame. If the columns are passed as manually constructed list, `aggregate` will use names like `Group.1` to identify the grouping variables, unless names are provided for the list elements.

As an example, suppose we wish to calculate the mean weight for observations in the `ChickWeight` data frame, broken down by the variables `Time` and `Diet`. Specifying the grouping variables as `ChickWeight[c('Time','Diet')]` will result in the grouping columns being properly labeled:

```
> cweights = > aggregate(ChickWeight$weight,
+                        ChickWeight[c('Time','Diet')],mean)
> head(cweights)
  Time Diet        x
1    0    1 41.40000
2    2    1 47.25000
3    4    1 56.47368
4    6    1 66.78947
5    8    1 79.68421
6   10    1 93.05263
```

Alternatively, a constructed list like

```
list(Time=ChickWeight$Time,Diet=ChickWeight$Diet)
```

could be used to achieve the same result.

To process a single vector based on the values of one or more grouping vectors, the `tapply` function can also be used. The returned value from `tapply` will be an array with as many dimensions as there were vectors that defined the groups. For example, the `PlantGrowth` dataset contains information about the weight of plants receiving one of three different treatments. To find the maximum weight for plants exposed to each of the treatments, we could use `tapply` as follows:

```
> maxweight = tapply(PlantGrowth$weight,PlantGrowth$group,max)
> maxweight
ctrl trt1 trt2
6.11 6.03 6.31
```

Since there was only one grouping factor, the results were returned in the form of a named vector. To convert this vector into a data frame, it can temporarily be converted into a table using `as.table`, and then passed to `as.data.frame`, since there is a special method for converting tables into data frames:

```
> as.data.frame(as.table(maxweight))
  Var1 Freq
1 ctrl 6.11
2 trt1 6.03
3 trt2 6.31
```

To use a name other than `Freq` in the data frame, `as.data.frame.table` can be called directly, using the `responseName=` argument:

```
> as.data.frame.table(as.table(maxweight),
                      responseName='MaxWeight')
  Var1 MaxWeight
1 ctrl      6.11
2 trt1      6.03
3 trt2      6.31
```

Unlike `aggregate`, `tapply` is not limited to returning scalars. For example, if we wanted the range of weights for each group in the `PlantGrowth` dataset, we could use

```
> ranges = tapply(PlantGrowth$weight,PlantGrowth$group,range)
> ranges
$ctrl
[1] 4.17 6.11

$trt1
[1] 3.59 6.03

$trt2
[1] 4.92 6.31
```

In this case. `tapply` returns a named array of vectors. Individual elements can be accessed in the usual way:

```
> ranges[[1]]
[1] 4.17 6.11
> ranges[['trt1']]
[1] 3.59 6.03
```

To convert values like this to data frames, the `dimnames` of the returned object can be combined with the values. When each element of the vector is of the same length, this operation is fairly straightforward, but the problem becomes difficult when the return values are of different lengths. In the current example, we can convert the values to a numeric matrix, and then form a data frame by combining the matrix with the `dimnames`:

```
> data.frame(group=dimnames(ranges)[[1]],
+            matrix(unlist(ranges),ncol=2,byrow=TRUE))
  group   X1   X2
1  ctrl 4.17 6.11
2  trt1 3.59 6.03
3  trt2 4.92 6.31
```

`data.frame` was used here instead of `cbind` to prevent the numeric values from being coerced to character values when they were combined with the levels of the grouping variable.

When more than one grouping variable is used with `tapply`, and the return value from the function used is not a scalar, the returned object is somewhat more difficult to interpret. For example, the `CO2` dataset contains information about the uptake of carbon dioxide by different types of plants exposed to different treatments. Suppose we were interested in the range of CO_2 uptake for plants of each type and treatment. We can call `tapply` as follows:

```
> ranges1 = tapply(CO2$uptake,CO2[c('Type','Treatment')],range)
> ranges1
             Treatment
Type          nonchilled chilled
  Quebec      Numeric,2  Numeric,2
  Mississippi Numeric,2  Numeric,2
```

The returned value is a matrix of lists, which explains the unusual form of the output when we display the object. Individual elements can still be accessed as expected:

```
> ranges[['Quebec','chilled']]
[1]  9.3 42.4
```

Such objects can be converted to data frames by applying `expand.grid` (see Section 2.8.1) to the dimnames before combining them with the values:

```
> data.frame(expand.grid(dimnames(ranges1)),
+            matrix(unlist(ranges1),byrow=TRUE,ncol=2))
         Type  Treatment   X1   X2
1      Quebec nonchilled 13.6 45.5
2 Mississippi nonchilled 10.6 35.5
3      Quebec    chilled  9.3 42.4
4 Mississippi    chilled  7.7 22.2
```

The function argument to `tapply` is not required; calling `tapply` without a function will return a vector of indices which can be used as a subscript to the array of values that `tapply` produces when a function is provided. For example, suppose we wish to subtract the median value of the `uptake` variable in the `CO2` data frame, where the median is calculated separately for each `Type`/`Treatment` combination. The first step is calculating the medians for each group using `tapply`:

```
> meds = tapply(CO2$uptake,CO2[c('Type','Treatment')],median)
```

Next, the indices are calculated using an identical call to `tapply` without a function, and they are used as a subscript to the median vector:

```
> inds = tapply(CO2$uptake,CO2[c('Type','Treatment')])
> inds
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3
[31] 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[61] 2 2 2 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
> adj.uptake = CO2$uptake - meds[inds]
```

The `ave` function combines these two operations in a single function call:

```
> adj.uptake = CO2$uptake -
+       ave(CO2$uptake,CO2[c('Type','Treatment')],FUN=median)
```

Since `ave` can accept multiple grouping variables, the function to be used for summarization must be identified using `FUN=`. Thus, the previous example could have been carried out with the following statement:

```
> adj.uptake = CO2$uptake -
+       ave(CO2$uptake,CO2$Type,CO2$Treatment,FUN=median)
```

When more than a single vector needs to be processed, a variety of options is available. To put the problem into context, consider the task of finding the maximum eigenvalue of the correlation matrices of the four variables from the `iris` dataset, broken down by the species of the plant. One solution is to use the `split` function, which takes a data frame and a list of grouping variables and returns a list containing data frames representing the observations for each level of the grouping variables. Such a list can then be processed using `sapply` or `lapply` to provide the final result. When working with problems like this, the first step is usually defining a function to provide the required result for a single data frame. In this case, an appropriate function could be written as follows:

```
> maxeig = function(df)eigen(cor(df))$val[1]
```

Next, the numeric values in the data frame can be passed to `split` to provide a list of data frames for further processing:

```
> frames = split(iris[-5],iris[5])
```

Finally, this result can be passed to `sapply` along with the function to do the work:

```
> sapply(frames,maxeig)
    setosa versicolor  virginica
  2.058540   2.926341   2.454737
```

As always, these operations can be condensed to a single expression, although there is no great advantage in doing so.

```
> sapply(split(iris[-5],iris[5]),
+        function(df)eigen(cor(df))$val[1])
    setosa versicolor  virginica
  2.058540   2.926341   2.454737
```

A less direct, but sometimes useful solution involves passing a vector of row indices to `tapply` and modifying the function used to calculate the maximum eigenvalue to operate on selected rows of the data:

```
> tapply(1:nrow(iris),iris['Species'],
+        function(ind,data)eigen(cor(data[ind,-5]))$val[1],
        data=iris)
Species
    setosa versicolor  virginica
  2.058540   2.926341   2.454737
```

Finally, the `by` function can be used. This generalizes the idea of `tapply` to operate on entire data frames broken down by a list of grouping variables. Thus, the first argument to `by` is a data frame, and the remaining arguments are similar to those of `tapply`. For the eigenvalue problem, a solution using `by` is as follows:

```
> max.e = by(iris,iris$Species,
+                 function(df)eigen(cor(df[-5]))$val[1])
> max.e
iris$Species: setosa
[1] 2.058540
------------------------------------------------------------
iris$Species: versicolor
[1] 2.926341
------------------------------------------------------------
iris$Species: virginica
[1] 2.454737
```

In this case, `by` returned a scalar, so the result can be converted to a data frame by using a combination of `as.table` and `as.data.frame`:

```
> as.data.frame(as.table(max.e))
  iris.Species     Freq
1       setosa 2.058540
2   versicolor 2.926341
3    virginica 2.454737
```

When there are multiple variables describing the groups to be processed, the result from `by` needs additional processing to get it in the form of a data frame. Consider again the `CO2` dataset. Suppose we wish to find the number of observations, mean, and standard deviation of the variable `uptake`, broken down by `Type` and `Treatment` combinations. First, a simple function to return the required values is written. By putting together the values with `data.frame`

instead of `c`, we insure that the mode of the numeric results will be preserved after we combine them with the level information for the grouping variables:

```
> sumfun = function(x)data.frame(n=length(x$uptake),
+                       mean=mean(x$uptake),sd=sd(x$uptake))
> bb = by(CO2,CO2[c('Type','Treatment')],sumfun)
> bb
Type: Quebec
Treatment: nonchilled
   n     mean      sd
1 21 35.33333 9.59637
------------------------------------------------------------
Type: Mississippi
Treatment: nonchilled
   n     mean       sd
1 21 25.95238 7.402136
------------------------------------------------------------
Type: Quebec
Treatment: chilled
   n     mean       sd
1 21 31.75238 9.644823
------------------------------------------------------------
Type: Mississippi
Treatment: chilled
   n     mean       sd
1 21 15.81429 4.058976
```

Each of the rows returned by the `by` function is in the form that we would like for a data frame containing these results, so it would be natural to use `rbind` to convert this result to a data frame; however, it is tedious to pass each row to the `rbind` function individually. In cases like this, the `do.call` function, first introduced in Section 6.5, can usually generalize the operation so that it will be carried out properly regardless of how many elements need to be processed. Recall that `do.call` takes a list of arguments and passes them to a function as if they were the argument list for the function call. In this example, the call to `do.call` is as follows:

```
> do.call(rbind,bb)
    n     mean       sd
1  21 35.33333 9.596371
11 21 25.95238 7.402136
12 21 31.75238 9.644823
13 21 15.81429 4.058976
```

With two grouping variables, the names and levels of the grouping factors are not present in the result. This can be remedied by combining a call to

`expand.grid` with the previous result. Since all the parts being combined are data frames, they can be safely combined using `cbind`:

```
> cbind(expand.grid(dimnames(bb)),do.call(rbind,bb))
         Type  Treatment  n     mean       sd
1      Quebec nonchilled 21 35.33333 9.596371
2 Mississippi nonchilled 21 25.95238 7.402136
3      Quebec    chilled 21 31.75238 9.644823
4 Mississippi    chilled 21 15.81429 4.058976
```

8.6 The `reshape` Package

An alternative approach to aggregation is provided by the `reshape` package, available from CRAN. The functions in this package provide a unified approach to aggregation, based on an extended formula notation. The core idea behind the `reshape` package is to create a "melted" version of a dataset (through the `melt` function), which can then be "cast" (with the `cast` function) into an object with the desired orientation. To melt a data frame, list, or array into the appropriate melted form, it is first necessary to divide the variables into id variables and measure or analysis variables; this should generally be obvious from the nature of the data. By default, `melt` treats factor and integer variables as id variables, and the remaining variables as analysis variables; if your data is structured according to this convention, no additional information needs to be provided to `melt`. Otherwise, the `id.var=` or `measure.var=` arguments can be used; if you specify one, it will assume all the other variables are of the other type. Once a dataset is melted, it can be cast into a variety of forms.

As a simple example, consider a dataset formed from the `state.x77` data frame, combined with the `state.region` variable:

```
> states = data.frame(state.x77,state=row.names(state.x77),
+                     region=state.region,row.names=1:50)
```

The `state` and `region` variables are stored as factors, so they will be automatically recognized as id variables when we melt the data:

```
> library(reshape)
> mstates = melt(states)
Using state, region as id variables
```

Notice that `melt` displays the names of variables that have been automatically assigned as id variables. The basic `melt`ing operation preserves the id variables, and converts the measured variables into two columns named `variable` (which identifies which variable is being measured) and `value` (which contains the actual values). You can use a name other than `variable` by specifying a `variable_name=` argument to `melt`.

The left-hand side of the formula passed to `cast` represents the variable(s) which will appear in the columns of the result, and the right-hand side describes the variables which will appear in the rows. Formulas used by `cast` can include a single dot (`.`) to represent an overall summary, or three dots `...` to represent all variables not otherwise included in the formula. In the simplest case, we can reproduce the original dataset with a formula like "`... ~ variable`".

When used for aggregation, an aggregation function should be supplied; if not it defers to using `length`. Suppose we wish to find the mean for each variable, broken down by region, with the regions appearing as a column in the output data frame:

```
> cast(mstates,region~variable,mean)
         region Population   Income Illiteracy Life.Exp
1     Northeast   5495.111 4570.222   1.000000 71.26444
2         South   4208.125 4011.938   1.737500 69.70625
3 North Central   4803.000 4611.083   0.700000 71.76667
4          West   2915.308 4702.615   1.023077 71.23462
     Murder  HS.Grad    Frost      Area
1  4.722222 53.96667 132.7778  18141.00
2 10.581250 44.34375  64.6250  54605.12
3  5.275000 54.51667 138.8333  62652.00
4  7.215385 62.00000 102.1538 134463.00
```

If we wanted a separate row for each `variable` instead of each `region`, we can reverse the role of those variables in the formula:

```
> cast(mstates,variable~region,mean)
    variable    Northeast       South North Central
1 Population  5495.111111  4208.12500    4803.00000
2     Income  4570.222222  4011.93750    4611.08333
3 Illiteracy     1.000000     1.73750       0.70000
4   Life.Exp    71.264444    69.70625      71.76667
5     Murder     4.722222    10.58125       5.27500
6    HS.Grad    53.966667    44.34375      54.51667
7      Frost   132.777778    64.62500     138.83333
8       Area 18141.000000 54605.12500   62652.00000
          West
1 2.915308e+03
2 4.702615e+03
3 1.023077e+00
4 7.123462e+01
5 7.215385e+00
6 6.200000e+01
7 1.021538e+02
8 1.344630e+05
```

To limit the variables that are used, we can use the `subset=` argument of `cast`. Since this argument uses the melted data, we need to refer to the variable named `variable`:

```
> cast(mstates,region~variable,mean,
+              subset=variable %in% c('Population','Life.Exp'))
         region Population Life.Exp
1     Northeast   5495.111 71.26444
2         South   4208.125 69.70625
3 North Central   4803.000 71.76667
4          West   2915.308 71.23462
```

Unlike the `aggregate` function which does not accept functions which return vectors of values, `cast` allows such functions, and uses the names of the returned vector to form new variable names in its output. Alternatively, a list of functions can be provided. Suppose we wish to calculate the mean, median, and standard deviations for `Population` and `Lif.Exp` in the `states` data frame. Since built-in functions exist for each statistic, they can be passed to `cast` as a list: First, we can calculate these quantities for the entire dataset:

```
> cast(mstates,.~variable,c(mean,median,sd),
+              subset=variable %in% c('Population','Life.Exp'))
  value Population_mean Population_median Population_sd
1 (all)         4246.42            2838.5      4464.491
  Life.Exp_mean Life.Exp_median Life.Exp_sd
1       70.8786          70.675    1.342394
```

Since `variable` was specified on the right-hand side of the tilde, all of the statistics for all of the variables are listed in a single row. A more familiar form would have the variables listed in a column, once again achieved by reversing the roles of the variables in the formula:

```
> cast(mstates,variable~.,c(mean,median,sd),
+      subset=variable %in% c('Population','Life.Exp'))
    variable      mean   median          sd
1 Population 4246.4200 2838.500 4464.491433
2   Life.Exp   70.8786   70.675    1.342394
```

To aggregate using a grouping variable, the period in the formula can be replaced by the grouping variable, in this case `region`:

```
> cast(mstates,region~variable,c(mean,median,sd),
+              subset=variable %in% c('Population','Life.Exp'))
         region Population_mean Population_median Population_sd
1     Northeast        5495.111            3100.0      6079.565
2         South        4208.125            3710.5      2779.508
3 North Central        4803.000            4255.0      3702.828
4          West        2915.308            1144.0      5578.607
  Life.Exp_mean Life.Exp_median Life.Exp_sd
```

```
1      71.26444           71.23   0.7438769
2      69.70625           70.07   1.0221994
3      71.76667           72.28   1.0367285
4      71.23462           71.71   1.3519715
```

If the roles of `region` and `variable` were reversed, there would be one variable for each combination of `region` and `mean`, `median`, and `sd`, which might not be convenient for display or further manipulation. To provide added flexibility, the vertical bar (`|`) can be used to cause `cast` to produce a list instead of a data frame. To create a list with a separate data summary for each region, we can specify `region` after the vertical bar, and replace it with a period in the formula:

```
> cast(mstates,variable~.|region,
+              c(mean,median,sd),
+              subset=variable%in%c('Population','Life.Exp'))
$Northeast
    variable       mean  median           sd
1 Population 5495.11111 3100.00 6079.5651457
2   Life.Exp   71.26444   71.23    0.7438769

$South
    variable       mean  median          sd
1 Population 4208.12500 3710.50 2779.508251
2   Life.Exp   69.70625   70.07    1.022199

$`North Central`
    variable       mean  median          sd
1 Population 4803.00000 4255.00 3702.827593
2   Life.Exp   71.76667   72.28    1.036729

$West
    variable       mean  median          sd
1 Population 2915.30769 1144.00 5578.607015
2   Life.Exp   71.23462   71.71    1.351971
```

Note that this creates a separate list element for each region, and that the contents of these elements are similar to those created with the formula "`variable ~ .`" in a previous example.

The principles in the previous example extend readily to the case with more than one id variable. Consider once again the `ChickWeight` data frame. The variables in this dataset are `weight`, `Time`, `Chick`, and `Diet`. The last three variables represent id variables, with `weight` being the only measure variable. Since `Time` is stored as a numeric variable, it is necessary to explicitly provide either the id or measure variables to the `melt` function:

```
> mChick = melt(ChickWeight,measure.var='weight')
```

To create a data frame with the median value of `weight` for each level of `Diet` and `Time`, the following call to `cast` can be used:

```
> head(cast(mChick,Diet + Time ~ variable,median))
  Diet Time weight
1    1    0     41
2    1    2     49
3    1    4     56
4    1    6     67
5    1    8     79
6    1   10     93
```

Notice that the variable specified last on the left-hand side (`Time`) is the one that varies the fastest.

To create a separate column for the median at each time, `Time` can be moved to the right-hand side of the formula:

```
> cast(mChick,Diet ~ Time + variable,mean)
  Diet  0_weight  2_weight  4_weight  6_weight  8_weight
1    1      41.4     47.25  56.47368  66.78947  79.68421
2    2      40.7     49.40  59.80000  75.40000  91.70000
3    3      40.8     50.40  62.20000  77.90000  98.40000
4    4      41.0     51.80  64.50000  83.90000 105.60000
  10_weight 12_weight 14_weight 16_weight 18_weight 20_weight
1  93.05263  108.5263  123.3889  144.6471  158.9412  170.4118
2 108.50000  131.3000  141.9000  164.7000  187.7000  205.6000
3 117.10000  144.4000  164.5000  197.4000  233.1000  258.9000
4 126.00000  151.4000  161.8000  182.0000  202.9000  233.8889
  21_weight
1  177.7500
2  214.7000
3  270.3000
4  238.5556
```

To create a list, with one element for each `Diet`, and the median of `weight` for each `Time`, use the vertical bar as follows:

```
> cast(mChick,Time ~ variable|Diet,mean)
$`1`
   Time    weight
1     0  41.40000
2     2  47.25000
3     4  56.47368
4     6  66.78947
5     8  79.68421
6    10  93.05263

     .  .  .
```

```
$`4`
   Time   weight
1     0  41.0000
2     2  51.8000
3     4  64.5000
4     6  83.9000
5     8 105.6000
6    10 126.0000

      .   .   .
```

In the previous example there were valid values for each combination of the id variables. If this is not the case, the default behavior of `cast` is to only include combinations actually encountered in the data. To include all possible combinations, use the `add.missing=TRUE` argument. For example, suppose we remove one combination of `Diet` and `Time` from `ChickWeight`:

```
> xChickWeight = subset(ChickWeight,
+                !(Diet == 1 & Time == 4))
> mxChick = melt(xChickWeight,measure.var='weight')
> head(cast(mxChick,Diet + Time ~ variable,median))
  Diet Time weight
1    1    0     41
2    1    2     49
3    1    6     67
4    1    8     79
5    1   10     93
6    1   12    106
```

By using `add.missing=TRUE`, observations for the missing combinations will be created, with a missing value for the analysis variable:

```
> head(cast(mxChick,Diet + Time ~ variable,median,
+           add.missing=TRUE))
  Diet Time weight
1    1    0     41
2    1    2     49
3    1    4     NA
4    1    6     67
5    1    8     79
6    1   10     93
```

In each of the preceding examples, the dataset was first `melt`ed, then repeated calls to `cast` were carried out. If only a single call to cast is needed, the `recast` function combines the melt and cast steps into a single call:

```
> head(recast(xChickWeight,measure.var='weight',
```

```
+             Diet + Time ~ variable,median,
+             add.missing=TRUE))
  Diet Time weight
1    1    0     41
2    1    2     49
3    1    4     NA
4    1    6     67
5    1    8     79
6    1   10     93
```

8.7 Loops in R

In previous sections, the apply family of functions (and associated wrappers) has been presented as the first choice for most repetitive tasks, such as operating on each element of a list, or performing a computation for nonoverlapping subgroups of the data. The major factor in this decision has to do with the simplicity of the functions, as well as their ability to properly use any names which have been assigned to their input arguments. But this way of programming may be awkward and unfamiliar, and many people would like to leverage their knowledge of other programming languages into R by using more familiar programming constructs like loops. An examination of some of the apply-style functions' source code will show that these functions internally use loops to actually get their work done, so arguments against loops based solely on efficiency do not carry much weight. The real problem with loops is that there are some very intuitive operations that may be implemented with loops that turn out to be extremely inefficient in R. In this and the following sections, we'll access the efficiency of different approaches to common problems with the use of the `system.time` function. This function accepts any valid R expression, and returns a vector of length five, containing the user CPU time, the system CPU time, the elapsed time, and the user and system times from any subprocesses. The first value shown, user CPU, is usually the most useful measure of efficiency, and will vary less than the other values when the same task is repeated several times. Since the argument handling in functions uses equal signs to identify keywords, the one restriction when using `system.time` is that assignment statements which are to be timed must use the "gets" form of the assignment operator, namely, `<-` instead of the equal sign.

Before looking at the cases to avoid, let's consider a simple example: finding the mean of each column of a matrix. This problem is so common that the `rowMeans` function is provided for an extremely efficient solution:

```
> dat = matrix(rnorm(1000000),10000,100)
> system.time(mns <- rowMeans(dat))
[1] 0.008 0.000 0.010 0.000 0.000
```

Another solution is to use `apply`:

```
> system.time(mns <- apply(dat,2,mean))
[1] 0.032 0.020 0.056 0.000 0.000
```

Next, we can use a loop to calculate the mean of each column separately. Notice that in this case, we need to initialize the result vector `mns` to accommodate the answer:

```
> system.time({m <- ncol(dat)
+              for(i in 1:m)mns[i] <- mean(dat[,i])})
[1] 0.032 0.004 0.036 0.000 0.000
```

There really isn't that much of a difference in execution time (the loop uses slightly less system time). The main advantage of `apply` in this case is that it eliminates the need to worry about the result vector, and, if the matrix were named, those names would be passed on to the result.

Keep in mind that the previous example still took advantage of vectorization: each column mean was calculated from a single call to `mean`. It is almost always a mistake to loop over each element of a matrix. Consider the following function, which calculates the mean of each column of the matrix by adding together every element and then dividing by the column length:

```
> slowmean = function(dat){
+   n = dim(dat)[1]
+   m = dim(dat)[2]
+   mns = numeric(m)
+   for(i in 1:n){
+      sum = 0;
+      for(j in 1:m)sum = sum + dat[j]
+      mns[i] = sum / n
+   }
+   return(mns)
+}
> system.time(mns <- slowmean(dat))
[1] 2.100 0.000 2.097 0.000 0.000
```

Without any vectorization, the computation is much slower than the other solutions. This illustrates that unless some kind of vectorization is used, computations in R will be very slow.

Before leaving this problem, it should be mentioned that, for any given problem, there may be unique solutions available. For example, the mean of each column of a matrix can be calculated directly using matrix expressions as follows:

```
> system.time({m = dim(dat)[1];mns = rep(1,m) %*% dat / m})
[1] 0.020 0.000 0.021 0.000 0.000
```

This represents an improvement over the apply and loop-based solutions, but is still not as efficient as the `colMeans` solution.

This illustrates that loops, in and of themselves, are not necessarily inefficient in R, but they should certainly take advantage of any vectorization possible to keep them competitive with other techniques.

To understand the kinds of loops which cause problems in R, it's worthwhile to recall how matrices are stored in R, namely, as a one-dimensional vector, with the columns of the matrix "stacked" on top of each other. A very common operation is to build up a matrix iteratively, by starting with an empty matrix, and using the `rbind` function to grow the matrix one row at a time. There are two problems with this approach. First, the size of the matrix changes at each iteration, requiring additional time to be spent in memory allocations. More importantly, since adding a row changes the size of each column in the matrix, all of the matrix elements need to be rearranged in memory each time a new row is added. These repeated memory allocations and rearrangements very quickly take their toll on the efficiency of a program.

Consider the trivial task of creating a matrix, each of whose rows represent the numbers from 1 to 100. Because of recycling rules, this can be achieved as follows:

```
> system.time(m <- matrix(1:100,10000,100,byrow=TRUE))
[1] 0.022 0.003 0.025 0.000 0.000
```

Performing the same operation by incrementally building the matrix is much slower:

```
> buildrow = function(){
+    res = NULL
+    for(i in 1:10000)res = rbind(res,1:100)
+    res
+ }
> system.time(buildrow())
[1] 239.236  21.446 260.707   0.000   0.000
```

Two forces are slowing the computation: first, the size of `res` is changing each time a new row is added to the matrix, causing R to reallocate memory at each iteration. In addition, since R stores its matrices internally by columns, the addition of a row to the matrix means that every column in the matrix needs to be extended, resulting in large amounts of data being moved around in memory. By this reasoning, it would be faster to build the matrix by columns of equal size, since less rearrangement of the data will be necessary:

```
> buildcol = function(){
+    res = NULL
+    for(i in 1:10000)res = cbind(res,1:100)
+    t(res)
+ }
> system.time(buildcol())
[1] 142.666  20.596 163.289   0.000   0.000
```

While this does represent a speedup, it is still far from an optimal solution. What makes the first technique so fast is that when the `matrix` function is used, the size of the result can be determined before the data is generated. We can provide the same advantage to a loop-based solution as follows:

```
> buildrow1 = function(){
+    res = matrix(0,10000,100)
+    for(i in 1:10000)res[i,] = 1:100
+    res
+ }
> system.time(buildrow1())
[1] 0.242 0.015 0.257 0.000 0.000
```

Even if we didn't know how many rows the matrix would contain, it would still be faster to allocate more rows than we need, and then truncate the matrix at the end. For example, let's include only 50% of the rows by checking the value of a random number before adding that row to the output matrix. First, we'll start with a NULL matrix:

```
> somerow1 = function(){
+   res = NULL
+   for(i in 1:10000)if(runif(1) < .5)res = rbind(res,1:100)
+   res
+ }
> system.time(somerow1())
[1] 51.007  6.062 57.125  0.000  0.000
```

Next, we'll allocate a matrix large enough to hold all the rows, then truncate it at the end:

```
> somerow2 = function(){
+   res = matrix(0,10000,100)
+   k = 0
+   for(i in 1:10000)if(runif(1) < .5){
+        k = k + 1
+        res[k,] = 1:100
+   }
+   res[1:k,]
+ }
> system.time(somerow2())
[1] 0.376 0.027 0.404 0.000 0.000
```

Provided there is enough memory for the initial allocation, creating a sufficiently large matrix before beginning to build it will generally be much faster than repeatedly calling `rbind`.

If a situation arises where it is difficult or impossible to allocate an appropriate matrix before building the rows, we can take advantage of the fact that lists in R are stored very differently than matrices. In particular, the

memory used by list elements does not have to be contiguous, which means that adding elements to a list doesn't require as much manipulation of data within memory as the corresponding operation on a matrix. The strategy is to build a list of the rows that will eventually become the matrix, and then use `do.call` to pass all of the rows to `rbind` in a single operation:

```
> somerow3 = function(){
+    res = list()
+    for(i in 1:10000)if(runif(1) < .5)res = c(res,list(1:100))
+    do.call(rbind,res)
+ }
> system.time(somerow3())
[1] 33.308  0.247 33.575  0.000  0.000
```

While nowhere near as fast as more optimal methods, this technique may prove useful in those situations where the size of the final result may be difficult to determine.