LAB100

Week 16: Arrays and Lists

This workshop introduces two additional types of object in R and gives an example of a function that implements a simple sorting algorithm. Upon completing this workshop you should be able to:

- Construct arrays and lists in R
- Use arrays in conjuction with for loops to store data
- Construct functions that return a list as output
- Understand the Bubble sort algorithm

Remember that you need to change the working directory and open a new R script at the start of each workshop.

1 Arrays

In Weeks 13 and 14 we looked at matrices. Aside from their usage in linear algebra operations, matrices in R are a convenient way of storing data. In this respect, we can think of a matrix as a special case of a more general class of **arrays**. Whereas matrices have two dimensions (rows and columns), arrays can have an arbitrary number of dimensions.

An array can be assigned in R using the command `array`. This takes two main arguments; `data` specifies a vector of data, `dim` specifies a vector containing the dimensions of the array. We can specify a matrix using the `array` command, for instance

```
mat <- array(c(1,2,3,1,2,3),dim=c(3,2))
```

creates a (3×2) matrix and is equivalent to either of the following methods that we saw in Week 13:

```
mat <- matrix(c(1,2,3,1,2,3),3,2,byrow=FALSE)
mat <- cbind(c(1,2,3),c(1,2,3))
```

However, `array` allows us to make objects with a higher number of dimensions. For instance,

```
arr <- array(c(1,2,3,4,1,2,3,4),dim=c(2,2,2))
```

creates a $(2 \times 2 \times 2)$ array. One way of thinking about this object is as two (2×2) matrices stacked on top of one another. As with matrices, we can use the square bracket subscripting to retrieve elements or sections of the array. For instance

```
arr[1,2,1]
```

retrieves the (1,2,1) element of the three-dimensional array. If we subscript with blank values then we retrieve the whole of that dimension, for instance

```
arr[,,1]
```

will return a (2×2) matrix.

As an example, suppose we have a $\mathbb{R}^3 \to \mathbb{R}$ function, but we wish to find the minimum value attained on a finite set of input points. For instance, let

$$f(x, y, z) = (x - 5/2)^4 + (x + z - 6)^2 + (y - z + 1/3)^2$$

and suppose we want to find $\min f(x, y, z)$ where $x, y, z \in \{1, 2, 3, 4, 5\}$. One way of solving this problem is to compute the values of $f(x, y, z)$ and store them in a $(5 \times 5 \times 5)$ array. Note that we can first create an array of zeroes of the correct size using

```
f.store <- array(0,dim=c(5,5,5))
```

We can then use nested for loops to loop through all values

```
for (x in 1:5) {
    for (y in 1:5) {
       for (z in 1:5) {
           f.store[x,y,z] <- (x- 5/2)^4 + (x + z - 6)^2 + (y - z + 1/3)^2
       }
    }
}
```

We can then find the minimum value using

```
min(f.store)
```

we can also check whether there is a unique point at which this minimum attained by using `sum` to count how many elements of the array are equal to the minimum.

```
sum(f.store == min(f.store))
```

2 Lists

A `list` in R is an object that consists of a collection of objects which can be of different types. Lists provide a convenient way of grouping together objects. For instance, if we have a set of n vectors we want to group together, but the vectors are of various lengths, we would not be able to store these conveniently in a matrix, but we can store them as a list. Perhaps the most important use for lists are in situations where we have a function that computes several different quantities. Since a function only returns a single object, we need a list to group the different quantities.

For instance we can create a list consisting of a scalar, a vector and a matrix:

```
example.list <- list( 1, c(1,2), rbind(c(1,2,3),c(4,5,6)))
```

The individual components of the list can be retrieved in two ways. Firstly we can obtain the jth component in the list by using a double square bracket subscript, e.g.

```
example.list[[3]]
```

will return the 3rd component of the list, the matrix. However, in the same way as dataframes considered in the last workshop, when we create a list we can also associate names to the components:

```
example.list <- list( sc=1, vc=c(1,2), mx=rbind(c(1,2,3),c(4,5,6)))
```

and refer to them using a `$` subscript. For instance we can retrieve the vector using

```
example.list$vc
```

Note that if we want to refer to a particular value within a particular component of a list, we can either first retrieve the list component and store it as a new object, or we can combine two sets of subscripting together. For instance

```
example.list[[3]][2,3]
```

retrieves the (2,3) entry of the 3rd component of the list.

3 Sorting algorithm

A very common task in mathematical computing is to sort a vector of numbers into numerical order. The simplest, but by the no means the most algorithmically efficient way of doing this is via a *bubble sort*.

The bubble sort algorithm involves repeatedly looping through the vector of numbers, making pairwise comparisons between adjacent values. If the ith term of the vector is greater than the $(i+1)$th term, then the algorithm swaps these two entries. The algorithm terminates when it has done a full loop of the vector and finds no pairs out of order. The following code peforms the algorithm and also counts the number of comparisons and swaps made.

```
BubbleSort <- function( x ){
    if( !is.numeric(x) | !is.vector(x) ) stop("x must be a numerical vector.")
    num.swaps <- 0
    num.compare <- 0
    if (length(x) > 1) {
        current.swaps <- -1
        while(current.swaps != 0){
            current.swaps <- 0
            for(i in 1:(length(x)-1)){
                num.compare <- num.compare + 1
                if(x[i] > x[i+1]){
                    x[c(i,i+1)]<-x[c(i+1,i)]
                    num.swaps <- num.swaps + 1
                    current.swaps <- current.swaps + 1
                }
            }
        }
    }
return(list(x=x, num.swaps=num.swaps, num.compare=num.compare))
}
```

This returns a `list`. The sorted object is stored in the component `$x`, and the number of swaps and number of comparisons are stored in `$num.swaps` and `$num.compare` respectively.

Copy the code for the function into RStudio. Test that your bubble sort function with the example that we first considered.

```
BubbleSort( c( 7, 4, 1, 9) )
```

Notice that it performs the same number of comparisons and swaps as in the figure. Now try a larger problem and compare the result with the `sort` command.

```
data <- c(27, 37, 57, 91, 20, 90, 94, 66, 63, 6)
BubbleSort( data )$x
sort( data )
```

Quiz 1: Bubble Sort 1

Use the Bubble Sort algorithm to order the sequence `x` defined by

```
x <- c(seq(12,28,by=4),seq(10,0,by=-1),seq(19,11,by=-2))
```

into ascending order. How many swaps are made in total?

Quiz 2: Bubble Sort 2

Adapt the Bubble Sort algorithm code so that it sorts into descending order. How many swaps are made in order to sort the vector `x` as defined in Question 1 into descending order?

Quiz 3: Bubble Sort 3

Further adapt the Bubble Sort algorithm code so that the function terminates after a pre-specified number of pairwise swaps have been carried out. Use your adapted function to start sorting the vector `y` into **descending** order

```
y <- rep(0,50)
y[seq(1,49,by=2)]<- 25:1
y[seq(2,50,by=2)]<- 5:29
```

What is the 20th value of the partially sorted vector after the algorithm has performed exactly 200 pairwise swaps?

Quiz 4: Arrays 1

Recall question 3 from Week 9 on loops. As in that question, suppose I have the following numbers of coins of different denominations:

Coin	2p	5p	10p	20p	50p
Frequency	15	5	7	5	2

Construct a $(16 \times 6 \times 8 \times 6 \times 3)$ array and use nested for loops to store the values of each combination of coins within the array.
Based on your stored array determine how many combinations of coins give a value of at least £1.

Quiz 5: Arrays 2

Using the array constructed from the previous question, use subscripting and the `sum` function to establish how many of the combinations have fewer than ten 2p's, more than three 5p's and a combined value of at least £1.