STATS 380

R Programming

Palindromes

An Extended Example: Palindromic Numbers

- A word is *palindrome* if does not change when its letters are reversed.
- Common examples are: *civic*, *level*, *rotator*, *rotor*, *kayak*, *racecar*.
- Phrases and sentences can also be palindromes (in these cases spaces, letter case and punctuation are ignored).
- Two famous examples are:

  Able was I ere I saw Elba.
  A man, a plan, a canal – Panama!

- An (integer) number is a palindrome if reversing its digits produces an identical value.
- The values *1*, *121*, *24642*, etc., are palindromes.

Checking for Palindromes

- There is a simple way to check whether a word is a palindrome:

  > Reverse the letters in the word and see if the result is the same as the original word.

- This can be done in steps.

  - Extract the letters from the word.
  - Reverse their order.
  - Paste them back together into a word.
  - Compare with the original word.

- This can be done with the functions `substring` and `paste`.

Palindromes Checking Code

```
> x = "foobar"

> substring(x, 1:nchar(x), 1:nchar(x))
[1] "f" "o" "o" "b" "a" "r"

> substring(x, nchar(x):1, nchar(x):1)
[1] "r" "a" "b" "o" "o" "f"

> paste(c("f", "o", "o", "b", "a", "r"),
        collapse = "")
[1] "foobar"

> paste(substring(x, nchar(x):1, nchar(x):1),
        collapse = "")
[1] "raboof"
```

A Palindrome Checking Function

```
> strrev =
      function(x)
      paste(substring(x, nchar(x):1, nchar(x):1),
            collapse = "")

> is.palindrome =
      function(x)
      x[1] == strrev(x[1])

> is.palindrome("foobar")
[1] FALSE

> is.palindrome("racecar")
[1] TRUE
```

Checking for Numeric Palindromes

- The `is.palindrome` function is designed to work on character strings but, because of the magic of automatic coercion, it also works on numbers.

```
> is.palindrome(123)
[1] FALSE

> is.palindrome(12321)
[1] TRUE
```

- This happens because the `substring` function converts its first argument into a string and because the comparison `x[1] == strrev(x[1])`, between a number and a character string, converts the number to a character string before the comparison is made.

Code for Checking Numerical Palindromes

- The following code is designed to check whether a number is a palindrome.

```
> revdigits =
      function(n)
      as.numeric(strrev(as.numeric(n)))

> is.palindrome =
      function(n)
      n[1] == revdigits(n[1])

> is.palindrome(123)
[1] FALSE

> is.palindrome(121)
[1] TRUE
```

The Palindromic Level of Numbers

- If a number is not a palindrome, we can try to convert it into one by adding it to the value obtained by reversing its digits.
$$19+91 \to 110$$
- If this does not produce a palindrome, the process can be repeated.
$$110+011 \to 121$$
- The *palindromic level* of a number is the number of times that digit reversing and addition must be carried out before a palindrome is produced.
- The palindromic level values such as 2 or 121 is 0.
- The palindromic level of 19 is 2 because two reversals are required to produce a palindrome.

Computing the Palindromic Level

Writing a function to compute the palindromic level of a number is relatively easy.

```
> plevel =
      function(n) {
          level = 0
          while(n != revdigits(n)) {
              level = level + 1
              n = n + revdigits(n)
          }
          level
      }

> plevel(19)
[1] 2
```

A Vectorised Function

- The `plevel` function will only work for a single number which is in conflict with the general R philosophy of making functions work for vectors.

- It is easy to vectorise the function using a `for` loop.

```
> palindromic.level =
      function(n) {
          levels = numeric(length(n))
          for(i in 1:length(n))
              levels[i] = plevel(n[i])
          levels
      }

> palindromic.level(1:20)
 [1] 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 2 1
```

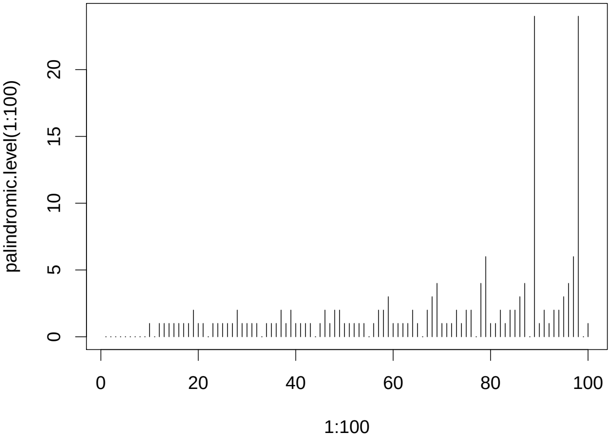
Palindromic Levels

- Using the function we've written it is easy to compute the palindromic levels of the first 100 integers.

```
> palindromic.level(1:100)
  [1]  0  0  0  0  0  0  0  0  0  1  0  1  1
 [14]  1  1  1  1  1  2  1  1  0  1  1  1  1
 [27]  1  2  1  1  1  1  0  1  1  1  2  1  2
 [40]  1  1  1  1  0  1  2  1  2  2  1  1  1
 [53]  1  1  0  1  2  2  3  1  1  1  1  2  1
 [66]  0  2  3  4  1  1  1  2  1  2  2  0  4
 [79]  6  1  1  2  1  2  2  3  4  0 24  1  2
 [92]  1  2  2  3  4  6 24  0  1
```

- Plotting the levels can be informative.

```
> plot(1:100, palindromic.level(1:100),
       type = "h")
```



palindromic.level(1:100)
0
5
10
15
20
0
20
40
60
80
100
1:100

Computational Problems

- Although the `palindromic.level` works fine for small values, it has clear problems with larger ones.

```
> palindromic.level(1:200)
Error in while (n != revdigits(n)) { :
  missing value where TRUE/FALSE needed
Calls: palindromic.level -> plevel
In addition: Warning message:
In revdigits(n) : NAs introduced by coercion
```

- There is clearly a problem in `revdigits`, at least some arguments.

Conversion of Numbers to Character Strings

- The process of reversing and adding can produce very big numbers.

- These are converted to scientific notation by `as.character`.

```
> x = 123456789012345678901234567890
> as.character(x)
[1] "1.23456789012346e+29"
```

- To ensure that scientific notation is not used, a different conversion function must be used.

```
> format(x, scientific = FALSE)
[1] "123456789012345677877719597056"
```

Computational Limits

- The process of reversing and adding can produce very large numbers.
- This is a problem because numbers bigger than 2^{53} may not be stored accurately in the computer.

```
> 2^53 == 2^53 + 1
[1] TRUE
```

- Because of this, we need to introduce a check to see whether numbers have grown too large and, if they have, to return a value that indicates this.
- The test can be implemented as follows:

```
> is.too.big =
      function(n) n >= 2^53
```

Program Modifications

- There are a number of changes which we can make to improve the `plevel` function.
  - Both the integer value and its reversed value must be checked to ensure that they are both accurate. If they are not, an `NA` can be returned.
  - The previous version of `plevel`, reversed the digits in the number twice. By saving the first value, the second reversal can be avoided.

Modified Code

```
> plevel =
      function(n) {
          level = 0
          while(n != (r = revdigits(n))) {
              if (is.too.big(n) ||
                  is.too.big(r)) {
                  level = NA
                  break
              }
              level = level + 1
              n = n + r
          }
          level
      }
```

Palindromic Levels

- The palindromic levels for the numbers between 101 and 200 can be computed as follows.

```
> palindromic.level(101:200)
 [1]  0  1  1  1  1  1  1  1  2  1  0  1  1
[14]  1  1  1  1  1  2  1  0  1  1  1  1  1
[27]  1  1  2  1  0  1  1  1  1  1  1  1  2
[40]  1  0  1  1  1  1  1  1  1  2  2  0  2
[53]  2  2  3  3  3  3  2  2  0  2  2  3  3
[66]  5 11  3  2  2  0  2  2  4  4  5 15  3
[79]  2  3  0  6  4  3  3  3 23  7  2  7  0
[92]  4  8  3  4 NA  7  5  3  1
```

- Notice that the level for 196 could not be computed.

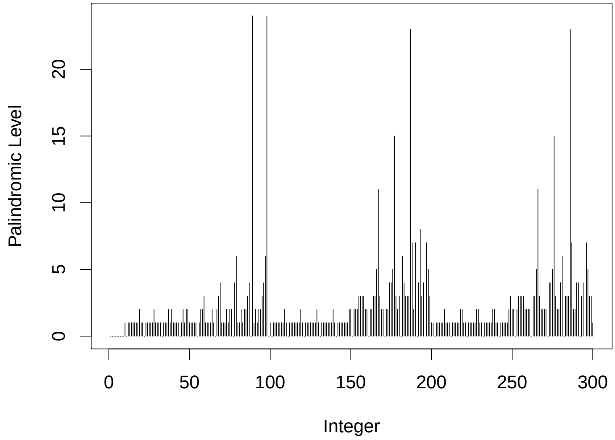
Lychrel Numbers

- Our program failed to compute the palindromic level of 196 because the values being computed during the computation became too big for R to handle accurately.
- In fact even with programs capable of working with numbers of up to 300,000,000 digits it has proved impossible to compute the palindromic level of 196.
- It has been conjectured that the palindromic level of 196 is infinite, but it is beyond the capabilities of present day mathematics to prove it.
- Numbers whose palindromic level is infinite are known as *Lychrel* numbers.

The Pattern of Palindromic Levels

- It is interesting to plot the pattern of palindromic levels for the first few hundred integers.

```
> plot(1:300, palindromic.level(1:300),
       type = "h",
       xlab = "Integer",
       ylab = "Palindromic Level")
```



Palindromic Level
0
5
10
15
20
0
50
100
150
200
250
300
Integer