# MAS115: R programming
# Lecture 5: Code complexity

March 9th 2021

# In the lab...

In the lab session, we'll look at some new data structures.

Factors and data frames are particularly useful in a statistical setting.

Lists are more general; they are useful for collecting together miscellaneous (usually named) objects into one bigger object. They will be particularly useful next week when we look in detail at writing functions in R. (We'll look at that briefly today, but won't need lists.)

# Coding efficiently
# Prime Numbers and Erastothenes' Sieve

# Finding all prime numbers smaller than $M$

A prime number is

- A natural number greater than 1
- Has no positive divisors other than 1 and itself

Suppose we want to find all the prime numbers smaller than $M$?

Question: How can we code this? Does it matter how we do it?

# Finding all prime numbers smaller than $M$

A prime number is

- A natural number greater than 1
- Has no positive divisors other than 1 and itself

Suppose we want to find all the prime numbers smaller than $M$?

Question: How can we code this? Does it matter how we do it?
Answer: Many ways and yes it does matter once $M$ becomes large

# A first attempt

Consider every number $n = 1, \ldots, M$ in turn

- Check if $n$ is prime by dividing by all numbers $i = 1, \ldots, n$
- If any factors found then composite, else prime

Pseudo-Code

```
INITIALIZE prime <- TRUE

FOR (i in 2:(n-1))
  IF (i divides n)
    THEN SET prime <- FALSE and break loop
  ENDIF
ENDFOR
RETURN prime
```

Need to modify slightly for the special case when $n = 2$

# Aside — R functions

In R (like most high level languages) it is easy to write your own functions. We'll look at this in detail next time; for now we just need the basic ideas.

- Lines of code can be grouped using { and }
- We use <- and the keyword `function` to create a function
- The keyword `function` is followed by the argument(s)
- The value returned by the function is specified by the keyword `return`
- User-defined functions are used just like built-in ones

# A first attempt — R code

I wrote some R code to check and number $n$ individually as follows

R Code 1

```
# Inputs: n number to check
# Outputs: TRUE if found to be prime
is.prime.slow <- function(n) {
  # Deal with special cases first
  if(n == 2) return(TRUE)
  # Otherwise try dividing it by all numbers up to n
  for(i in 2:(n-1)) {
    if((n %% i) == 0) {
      return(FALSE)
    }
  }
  return(TRUE)
}
```

# A first attempt - R code

We now have a function which can check whether an input number $n$ is prime so we can create a wrapper to check all the numbers $n = 2, \ldots, M$
Pseudo-Code

```
INITIALIZE primes as empty

FOR (i in 2:M)
  CHECK i primeness using is.prime.slow() function
  IF (i is prime)
    THEN ADD i to list of primes
  ENDIF
ENDFOR

RETURN primes
```

# A first attempt - R code

```
# Finds all the prime numbers less than a certain number
# Inputs: M find all prime numbers below this number
# Outputs: vector of all primes below M
findprimes1 <- function(M) {
  primes <- c()
  for(i in 2:M) {
    if(is.prime.slow(i)) primes <- c(primes, i)
  }
  primes
}
```

```
> findprimes1(20)
[1]  2  3  5  7 11 13 17 19
```

# Speed of our first attempt code

We can check how long our code takes to run by using the `system.time()` function

```
> system.time(p1 <- findprimes1(1000))
   user  system elapsed
   0.11    0.00    0.11
> system.time(p1 <- findprimes1(10000))
   user  system elapsed
  7.730   0.030   7.783
```

We haven't even chosen a very large $M$ and already it's taking ages. This is a real problem if we want to choose $M = 10,000,000$.

# Rough theory of algorithmic complexity - worst cases

is.prime.slow Consider the worst case where we have to divide each individual $n$ by all of the numbers $i = 2, \ldots, n-1$. Then the total number of operation we will have to perform is

$$\begin{aligned} \textit{Operations} &= 0 + 1 + 2 + 3 + \ldots + (M-2) = \sum_{n=2}^{M} (n-2) \\ &\approx \frac{M(M-1)}{2} \qquad [\text{summing a AP}] \\ &= \mathrm{O}\left(M^2\right) \qquad [\text{i.e. scales with } M^2] \end{aligned}$$

This is very slow

# An improvement - Trial 2

There is a lot of redundancy in our coding:

- To check if $n$ is prime we don't need to divide by all numbers $i = 1, \ldots, n$ but only need check for $i = 1, \ldots, \lfloor \sqrt{n} \rfloor$

Pseudo-Code

```
INITIALIZE prime <- TRUE

FOR (i in 2:sqrt(n))
  IF (i divides n)
    THEN SET prime <- FALSE and break loop
  ENDIF
ENDFOR
RETURN prime
```

Need to modify slightly for the special case when $n = 2$ and 3

# An improvement — R code

Can write this as a function in R too
R Code 2

```
# Inputs: n number to check
# Outputs: TRUE is found to be prime
is.prime.fast <- function(n) {
  # Deal with special cases first
  if((n == 2) || (n == 3)) return(TRUE)
  # Otherwise try dividing it by numbers
  for(i in 2:floor(sqrt(n))) {
    if((n %% i) == 0) {
      return(FALSE)
    }
  }
  return(TRUE)
}
```

# A second attempt - R code

We now have a more efficient function to check whether an input number $n$ is prime so we can use this in our wrapper instead

```
# Effort 2 at finding prime numbers less than M
# Inputs: M find all prime numbers below this number
# Outputs: vector of all primes below M
findprimes2 <- function(M) {
  primes <- c()
  for(i in 2:M) {
    if(is.prime.fast(i)) primes <- c(primes, i)
  }
  primes
}
```

```
> findprimes2(20)
[1]  2  3  5  7 11 13 17 19
```

# Speed of our second attempt code

We can check how much faster this new code runs

```
> system.time(p2 <- findprimes2(1000))
   user  system elapsed
  0.010   0.000   0.011
> system.time(p2 <- findprimes2(10000))
   user  system elapsed
  0.220   0.000   0.217
> system.time(p2 <- findprimes2(100000))
   user  system elapsed
  4.280   0.010   4.305
> system.time(p2 <- findprimes2(1000000))
   user  system elapsed
109.590   0.850 110.833
```

This is a lot faster but we're still going to struggle if we want a really large $M$ e.g. $M = 10,000,000$.

# Rough theory of algorithmic complexity - worst cases

is.prime.fast Here the worst case is we have to divide each individual $n$ by all of the numbers $i = 2, \ldots, \lfloor \sqrt{n} \rfloor$. Then the total number of operations would be

$$\begin{aligned}
\textit{Operations} &= \underbrace{\sqrt{2} + \sqrt{3}}_{\geq \sqrt{1}=1} + \underbrace{\sqrt{4} + \sqrt{5} + \sqrt{6} + \sqrt{7} + \sqrt{8}}_{\geq \sqrt{4}=2} + \sqrt{9} + \ldots + \sqrt{M} \\
&> \sum_{i=1}^{\sqrt{M}} i(2i+1) \qquad \left\{ \text{since} (i+1)^2 - i^2 = 2i + 1 \right\} \\
&\approx 2 \sum_{i=1}^{\sqrt{M}} i^2 \\
&= 2 \frac{2M^{3/2} + 3M + \sqrt{M}}{6} \\
&= \mathrm{O}\left( M^{\frac{3}{2}} \right) \qquad \text{Better but still not great}
\end{aligned}$$

# Erastothenes' Sieve

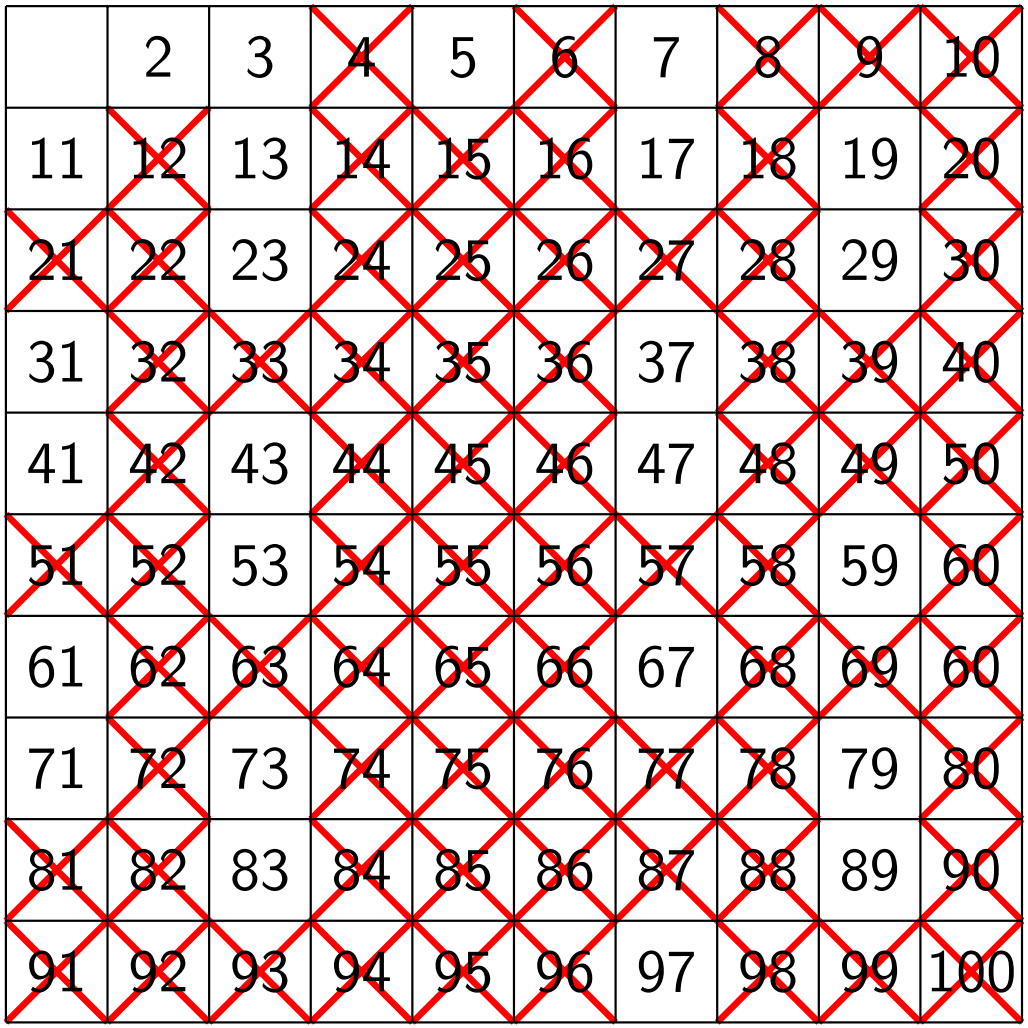
# Erastothenes' Sieve

To find all the prime numbers less than or equal to a given integer $M$:

1. Create a list of consecutive integers $(2, 3, 4, \ldots, M)$.
2. Initially, let $p$ equal 2, the first prime number.
3. Starting from $p$, mark each multiples of $p$: $2p$, $3p$, $4p$, etc.; note that some of them may have already been marked.
4. Find the first number greater than $p$ in the list that is not marked. If no such number, stop. Otherwise, let $p$ now equal this number (the next prime), and repeat from step 3.

As a refinement, it is sufficient to mark the numbers in step 3 starting from $p^2$, as all the smaller multiples of $p$ will have already been marked at that point. This means that the algorithm is allowed to terminate in step 4 when $p^2$ is greater than $M$.

# Erastothenes' Sieve - Illustration



# Erastothenes' Sieve - Pseudo-code

```
INITIALIZE unsieved as 2:M
INITIALIZE sieved as empty
INITIALIZE p = 0

WHILE (p^2 < M)
  SET p to be first element in unsieved
  REMOVE all multiples of p from unsieved
  APPEND p to sieved
ENDWHILE

SET primes = concatenate(sieved, unseived)
RETURN primes
```

# Erastothenes' Sieve - R code

```
primesieve <- function(M)
{
  unsieved <- 2:M
  sieved <- c()
  p <- 0
  while (p^2 < M) {
    p <- unsieved[1]
    unsieved <- unsieved[unsieved%%p != 0]
    sieved <- c(sieved, p)
  }
  return(c(sieved, unsieved))
}
```

```
> primesieve(20)
[1]  2  3  5  7 11 13 17 19
```

# Speed of Erastothenes' Sieve

How much faster is this algorithm?

```
> system.time(p3 <- primesieve(1000))
   user  system elapsed
  0.010   0.000   0.001
> system.time(p3 <- primesieve(10000))
   user  system elapsed
  0.000   0.000   0.002
> system.time(p3 <- primesieve(100000))
   user  system elapsed
  0.030   0.000   0.029
> system.time(p3 <- primesieve(1000000))
   user  system elapsed
  0.630   0.000   0.634
> system.time(p3 <- primesieve(10000000))
   user  system elapsed
 10.610   0.340  10.988
```

# Speed of the algorithms

How much difference did changing the way we coded our algorithm make?

| | Time (s) for $M$ | | | | |
|---|---|---|---|---|---|
| Method | 1000 | 10000 | 100000 | 1000000 | 10000000 |
| is.prime.slow | 0.11 | 7.783 | N/A | N/A | N/A |
| is.prime.fast | 0.011 | 0.217 | 4.305 | 110.8 | N/A |
| Erastothenes | 0.001 | 0.002 | 0.029 | 0.634 | 10.99 |

This a huge difference - it makes sense to think about writing your code efficiently

# Rough theory of algorithmic complexity - worst cases

Erastothenes' Sieve With Erastothenes' Sieve the number of operations we need to perform is

$$\mathrm{O}\left(M \log \log M\right)$$

This is much much faster than the other two algorithms

# Conclusion

- The way that you code your algorithms up can make a huge different to the speed that they run.
- Good to try and be as efficient as possible.
- More important to be correct (especially when starting out with programming)
- Efficiency often only makes a difference when you are working with really big numbers.