MT4113, Computing in Statistics

Lecture 2 - Algorithms to functions

19 September 2018

Algorithms

What is an algorithm?

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time

What is an algorithm?

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features:

What is an algorithm?

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features:
  - an ordered sequence

What is an algorithm?

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features:
  - an ordered sequence
  - unambiguous and well defined instructions - each instruction is clear, do-able, and can be done without difficulty

What is an algorithm?

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features:
  - an ordered sequence
  - unambiguous and well defined instructions - each instruction is clear, do-able, and can be done without difficulty
  - performs some task - algorithm needs to be *complete*, with nothing left out

What is an algorithm?

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features:
    - an ordered sequence
    - unambiguous and well defined instructions - each instruction is clear, do-able, and can be done without difficulty
    - performs some task - algorithm needs to be *complete*, with nothing left out
    - halts in finite time - i.e., the algorithm needs to terminate

Example algorithm

- Example algorithm

```
* Algorithm soft boiled egg *
Put water in pan.
When the water boils, turn over the egg timer.
When the timer has run out, turn off the heat.
Pour some cold water into the pan to cool the water.
Remove egg.
```

Example algorithm

- Example algorithm

```
* Algorithm soft boiled egg *
Put water in pan.
When the water boils, turn over the egg timer.
When the timer has run out, turn off the heat.
Pour some cold water into the pan to cool the water.
Remove egg.
```

- What is wrong with this algorithm?

Some elements of algorithms

Assignment and computation

- Assigning values to variables

Assignment and computation

- Assigning values to variables
  - `Let x := 1`

Assignment and computation

- Assigning values to variables
  - `Let x := 1`
- Computation

Assignment and computation

- Assigning values to variables
  - `Let x := 1`
- Computation
  - `x := x * 2`

Input and output

- Input

Input and output

- Input
  - Reading values from files

Input and output

- Input
  - Reading values from files
  - Passing values in at the start of the algorithm

Input and output

- Input
  - Reading values from files
  - Passing values in at the start of the algorithm
    - e.g., `Algorithm cakemix (sweet_tooth)`

Input and output

- Input
  - Reading values from files
  - Passing values in at the start of the algorithm
    - e.g., `Algorithm cakemix (sweet_tooth)`
    - `sweet_tooth` is a *parameter*

Input and output

- Input
  - Reading values from files
  - Passing values in at the start of the algorithm
    - e.g., `Algorithm cakemix (sweet_tooth)`
    - `sweet_tooth` is a *parameter*
- Output

Input and output

- Input
  - Reading values from files
  - Passing values in at the start of the algorithm
    - e.g., `Algorithm cakemix (sweet_tooth)`
    - `sweet_tooth` is a *parameter*
- Output
  - Printing results on screen

Input and output

- Input
  - Reading values from files
  - Passing values in at the start of the algorithm
    - e.g., `Algorithm cakemix (sweet_tooth)`
    - `sweet_tooth` is a *parameter*
- Output
  - Printing results on screen
  - Saving results to file

Control structures: conditional execution

- Conditional operations ask a true/false question and then select the next instruction based on the answer

Control structures: conditional execution

- Conditional operations ask a true/false question and then select the next instruction based on the answer
- Example:

```
* Algorithm cakemix (sweet_tooth) *
In a bowl mix together:
  8 oz butter
  4 medium eggs
  2tsp vanilla extract
  8 oz self raising flour
  if (sweet_tooth) then
    8 oz caster sugar
  else
    4 oz caster sugar
  end if
  ...
```

Control structures: iteration

- Iteration (looping): used to repeat a set of instructions

Control structures: iteration

- Iteration (looping): used to repeat a set of instructions
- **Number of iterations known at outset** - example:

```
* Algorithm scramble (n) *
Do these statements n times:
  take an egg out of the fridge;
  crack the egg's shell on the edge of the bowl;
  pull the egg apart above the bowl;
  let the contents of the egg fall into the bowl;
  put the egg shell into the bin.
Stir egg conetnts in the bowl.
Pour contents of bowl into the frying pan.
...
```

Control structures: iteration

- Iteration (looping): used to repeat a set of instructions
- **Number of iterations known at outset** - example:

```
* Algorithm scramble (n) *
Do these statements n times:
  take an egg out of the fridge;
  crack the egg's shell on the edge of the bowl;
  pull the egg apart above the bowl;
  let the contents of the egg fall into the bowl;
  put the egg shell into the bin.
Stir egg conetnts in the bowl.
Pour contents of bowl into the frying pan.
...
```

- Notice the way indenting has been used to group commands together. Numbering could also be used (1.1, 1.2, etc.)

- **Number of iterations not known at outset** (i.e., using a conditional operation to control the number of times a loop is executed).

- **Number of iterations not known at outset** (i.e., using a conditional operation to control the number of times a loop is executed).
- Example:

```
* Algorithm cup of water *
...
Do these statements until the water level is on the line:
  If the level is above the line pour a little water out;
  If the level is below the line pour a little water in;
...
```

- **Number of iterations not known at outset** (i.e., using a conditional operation to control the number of times a loop is executed).
- Example:

```
* Algorithm cup of water *
...
Do these statements until the water level is on the line:
  If the level is above the line pour a little water out;
  If the level is below the line pour a little water in;
...
```

- Warning - potential for an infinite loop!

- **Number of iterations not known at outset** (i.e., using a conditional operation to control the number of times a loop is executed).
- Example:

```
* Algorithm cup of water *
...
Do these statements until the water level is on the line:
  If the level is above the line pour a little water out;
  If the level is below the line pour a little water in;
...
```

- Warning - potential for an infinite loop!
  - Turn exact test into a "good enough" test – see Lecture 4

- **Number of iterations not known at outset** (i.e., using a conditional operation to control the number of times a loop is executed).
- Example:

```
* Algorithm cup of water *
...
Do these statements until the water level is on the line:
  If the level is above the line pour a little water out;
  If the level is below the line pour a little water in;
...
```

- Warning - potential for an infinite loop!
  - Turn exact test into a "good enough" test – see Lecture 4
  - Add a stated limit on the number of iterations – see Optimziation lectures

Recursion

- Recursion: algorithms that call themselves.

Recursion

- Recursion: algorithms that call themselves.
- Example:

```
* Algorithm factorial (n) *
If n == 1 then factorial = 1
  Else factorial = n * factorial(n - 1)
```

Code and Pseudocode

What is code?

- Code: instructions in a computer language that implement an algorithm.

- Example: factorial program in R

```
my.factorial <- function(n) {
  if (n == 1) {
    return(1)
  } else {
    return(n * my.factorial (n-1))
  }
}
```

- Example: factorial program in R

```
my.factorial <- function(n) {
  if (n == 1) {
    return(1)
  } else {
    return(n * my.factorial (n-1))
  }
}
```

- What's wrong with this function?

What is pseudocode?

- Pseudocode: instructions in a generic, informal, unspecified computer language that specify an algorithm.

What is pseudocode?

- Pseudocode: instructions in a generic, informal, unspecified computer language that specify an algorithm.
- Example - `Algorithm factorial (n)`

What is pseudocode?

- Pseudocode: instructions in a generic, informal, unspecified computer language that specify an algorithm.
- Example - `Algorithm factorial (n)`
- Tip: for anything other than completely trivial tasks, write out pseudocode on paper before writing any code on the computer!

What is pseudocode?

- Pseudocode: instructions in a generic, informal, unspecified computer language that specify an algorithm.
- Example - `Algorithm factorial (n)`
- Tip: for anything other than completely trivial tasks, write out pseudocode on paper before writing any code on the computer!
  - (For more on this, see next lecture)

Control structures in R

Conditional execution

- Single condition

```
if (x < 10) y <- 44
```

- Multiple conditions

```
if ((x < 10)|(y >= 12)) z <- 44
```

```
if ((x < 10)&(y >= 12)) z <- 44
```

- Two alternative outcomes

```
if (x < 10) {
  y <- 44
} else {
  z <- 44
}
```

- Multiple alternative outcomes for the same variable

```
today <- "Wednesday"
event.4113 <- switch(today,
                    "Monday" = "lecture if week number is odd",
                    "Tuesday" = "frantic revision",
                    "Wednesday" = "thrilling lecture",
                    "Thursday" = "frantic revision",
                    "Friday" = "exciting practical")
print(paste(today, "will hold in store", event.4113))
```

```
## [1] "Wednesday will hold in store thrilling lecture"
```

- Multiple alternative outcomes via nested conditional statements - more flexible but less elegant

```
if (x < 10) {
  y <- 44
} else {
  if (p == 14) {
    z <- 44
  } else {
    y <- x + a
    today <- "Wednesday"
  }
}
```

Iteration (loops) - number of iterations known at outset

```
for (i in 1:10) {
  x[i] <- x[i] * pi
}
```

(Note - in many cases such loops can be vectorized.)

Iteration - number of iterations unknown at outset

While loop:

```
while (i < 27) {
  x[i] <- x[i] * pi
}
```

Repeat until loop:

```
repeat {
  x[i] <- x[i] * pi
  if (i >= 27) break
}
```

What is the difference between these?

Vectorization

- Many operations in R can be vectorized - these run much faster than loops

```
n<-1E8
x <- rnorm(n)
system.time(
  for (i in 1:n) {
    x[i] <- x[i] * pi
  }
)
```

```
## user system elapsed
## 14.18 0.03 14.33
```

```
n<-1E8
x <- rnorm(n)
system.time(
  x <- x * pi
)
```

```
## user system elapsed
## 0.16 0.05 0.21
```

The `apply` family of functions

- If data organized into data frames or matrices

```r
data(iris3)
apply(X = iris3, MARGIN = 2, FUN = mean)
```

```
## Sepal L. Sepal W. Petal L. Petal W.
## 5.843333 3.057333 3.758000 1.199333
```

The `apply` family of functions

- If data organized into data frames or matrices
- You often need to iterate across colums or rows of the data

```
data(iris3)
apply(X = iris3, MARGIN = 2, FUN = mean)
```

```
## Sepal L. Sepal W. Petal L. Petal W.
## 5.843333 3.057333 3.758000 1.199333
```

The `apply` family of functions

- If data organized into data frames or matrices
- You often need to iterate across colums or rows of the data

```
data(iris3)
apply(X = iris3, MARGIN = 2, FUN = mean)
```

```
## Sepal L. Sepal W. Petal L. Petal W.
## 5.843333 3.057333 3.758000 1.199333
```

- There are several variants of `apply` - `sapply`, `lapply`, `tapply`, `mapply`, `%by%`

The `apply` family of functions

- If data organized into data frames or matrices
- You often need to iterate across colums or rows of the data

```
data(iris3)
apply(X = iris3, MARGIN = 2, FUN = mean)
```

```
## Sepal L. Sepal W. Petal L. Petal W.
## 5.843333 3.057333 3.758000 1.199333
```

- There are several variants of `apply` - `sapply`, `lapply`, `tapply`, `mapply`, `%by%`
- See, e.g., Matloff (2011) Sections 5.2 and 5.4 (esp 5.4.2)

The `apply` family of functions

- If data organized into data frames or matrices
- You often need to iterate across colums or rows of the data

```r
data(iris3)
apply(X = iris3, MARGIN = 2, FUN = mean)
```

```
## Sepal L. Sepal W. Petal L. Petal W.
## 5.843333 3.057333 3.758000 1.199333
```

- There are several variants of `apply` - `sapply`, `lapply`, `tapply`, `mapply`, `%by%`
- See, e.g., Matloff (2011) Sections 5.2 and 5.4 (esp 5.4.2)
- Vectorization in `R` takes some getting used to - but persistence pays off!

Modular programming

what is "modular programming"?

Splittling a program into discrete, reuseable blocks of code so that each block does a small amount.

Motivation

Convert `-99` to `NA` in a dataset, of which this is a fraction

```
##     a   b   c   d   e f
## 1  6   7   2   5   2 4
## 2  1   8 -99 -99 -99 6
## 3  1   5   9   5 -99 4
## 4  6 -99   9   6   1 1
## 5  9   8   6   1   8 9
## 6 10   7   9   9   5 5
```

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
df
```

```
##    a  b   c  d  e f
## 1  6  7   2  5  2 4
## 2  1  8 -99 NA NA 6
## 3  1  5   9  5 NA 4
## 4  6 NA   9  6  1 1
## 5  9  8   6  1  8 9
## 6 10  7   9  9  5 5
```

What happened?

- Functions are the R way to implement modular programming

```
fix.missing <- function(x) {
  x[x == -99] <- NA
  return(x)
}
```

```
df$a <- fix.missing(df$a)
df$b <- fix.missing(df$b)
df$c <- fix.missing(df$c)
df$d <- fix.missing(df$d)
df$e <- fix.missing(df$e)
df$f <- fix.missing(df$e)
```

- Functions are the R way to implement modular programming

```
fix.missing <- function(x) {
  x[x == -99] <- NA
  return(x)
}
```

```
df$a <- fix.missing(df$a)
df$b <- fix.missing(df$b)
df$c <- fix.missing(df$c)
df$d <- fix.missing(df$d)
df$e <- fix.missing(df$e)
df$f <- fix.missing(df$e)
```

- Still too much redundant code

- Functions are the R way to implement modular programming

```
fix.missing <- function(x) {
  x[x == -99] <- NA
  return(x)
}
```

```
df$a <- fix.missing(df$a)
df$b <- fix.missing(df$b)
df$c <- fix.missing(df$c)
df$d <- fix.missing(df$d)
df$e <- fix.missing(df$e)
df$f <- fix.missing(df$e)
```

- Still too much redundant code
  - DRY principle

- Functions are the R way to implement modular programming

```
fix.missing <- function(x) {
  x[x == -99] <- NA
  return(x)
}
```

```
df$a <- fix.missing(df$a)
df$b <- fix.missing(df$b)
df$c <- fix.missing(df$c)
df$d <- fix.missing(df$d)
df$e <- fix.missing(df$e)
df$f <- fix.missing(df$e)
```

- Still too much redundant code
  - DRY principle
    - Don't Repeat Yourself

- Make use of inherent vectorisation in R

```r
fix.missing(df)
```

```
##     a  b  c  d  e f
## 1   6  7  2  5  2 4
## 2   1  8 NA NA NA 6
## 3   1  5  9  5 NA 4
## 4   6 NA  9  6  1 1
## 5   9  8  6  1  8 9
## 6  10  7  9  9  5 5
```

Modules in R

- In R, modular programming is implemented through the use of **functions**

```
function.name <- function(arguments) {
  body
}
```

Modules in R

- In R, modular programming is implemented through the use of **functions**
- Syntax:

```
function.name <- function(arguments) {
  body
}
```

Example R function

- Exploratory data analysis function - shows 3 plots and returns a summary of data vector x

```
eda <- function (x) {
  par(mfrow = c(1 ,3))
  hist(x, probability = TRUE)
  lines(density(x))
  boxplot(x, horizontal = TRUE)
  rug(x)
  qqnorm(x)
  return(summary(x))
}
```

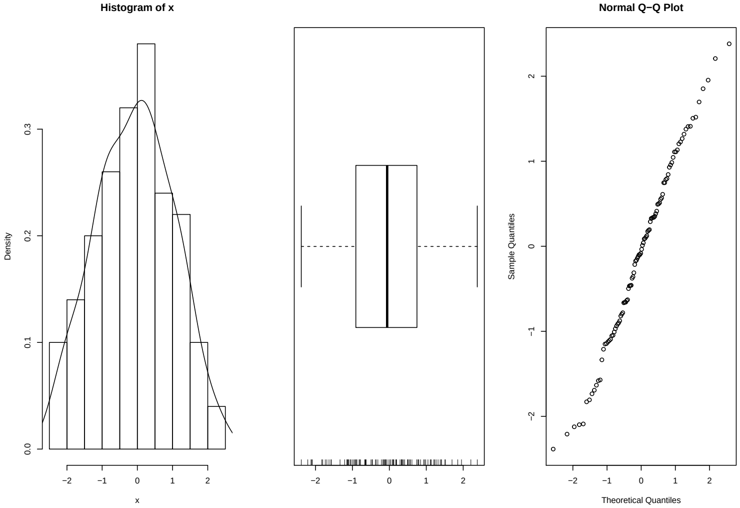
Example R function

- Exploratory data analysis function - shows 3 plots and returns a summary of data vector x

```
eda <- function (x) {
  par(mfrow = c(1 ,3))
  hist(x, probability = TRUE)
  lines(density(x))
  boxplot(x, horizontal = TRUE)
  rug(x)
  qqnorm(x)
  return(summary(x))
}
```

- (N.B. There are a few "bad" things about this function – see next lecture for what!)

```r
x <- rnorm(100)
eda(x)
```



Module interfaces - passing information in

- In general programmer-speak, the things you pass into modules are called “parameters”

```
print.and.multiply <- function (x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
```

Module interfaces - passing information in

- In general programmer-speak, the things you pass into modules are called “parameters”
- In ‘R they are called “arguments”

```
print.and.multiply <- function (x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
```

Module interfaces - passing information in

- In general programmer-speak, the things you pass into modules are called "parameters"
- In 'R they are called "arguments"
- E.g., in

```
print.and.multiply <- function (x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
```

Module interfaces - passing information in

- In general programmer-speak, the things you pass into modules are called “parameters”
- In ‘R they are called “arguments”
- E.g., in

```
print.and.multiply <- function (x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
```

- `x` and `y` are function arguments

```r
print.and.multiply <- function (x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
var.1 <- 10
var.2 <- 20
print.and.multiply(var.1, var.2)
```

```
## Inside function x= 10 y= 20
```

```
## [1] 200
```

- In computer languages, there are two ways to make parameters (arguments) available to functions:

- In computer languages, there are two ways to make parameters (arguments) available to functions:
  - passing by value

- In computer languages, there are two ways to make parameters (arguments) available to functions:
  - passing by value
  - passing by reference

Passing by value

- A copy is made of the value of each variable passed in to a function

Passing by value

- A copy is made of the value of each variable passed in to a function
- These copies are stored in a separate location in memory from the original variables

Passing by value

- A copy is made of the value of each variable passed in to a function
- These copies are stored in a separate location in memory from the original variables
- So, changes to the variable inside the function have no affect on its' value outside the function

```
print.and.multiply <- function (x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
var.1 <- 10
var.2 <- 20
print.and.multiply(var.1, var.2)
```

```
## Inside function x= 10 y= 20
```

```
## [1] 200
```

```
var.1
```

```
## [1] 10
```

- Aside - what would happen if you now typed `x`?

Passing by reference

- The memory location of the variable passed in is given to the function

Passing by reference

- The memory location of the variable passed in is given to the function
- Therefore any changes to the variable within the function affect its' value after the function has completed

Passing by reference

- The memory location of the variable passed in is given to the function
- Therefore any changes to the variable within the function affect its' value after the function has completed
- The following code will not work in R as it does not allow passing by reference:

```
print.and.multiply <- function (ByReference x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
var.1 <- 10
var.2 <- 20
print.and.multiply(var.1, var.2)
## [1] 200
var.1
## [1] 200
```

Pros and cons

- Passing by value

Pros and cons

- Passing by value
  - Safer - you can do what you like with the variable inside the function, and you won't affect what goes on outside the function

Pros and cons

- Passing by value
  - Safer - you can do what you like with the variable inside the function, and you won't affect what goes on outside the function
  - Inefficient - the computer has to make a copy of all variables - takes time and money

Pros and cons

- Passing by value
  - Safer - you can do what you like with the variable inside the function, and you won't affect what goes on outside the function
  - Inefficient - the computer has to make a copy of all variables - takes time and money
- Passing by reference

Pros and cons

- Passing by value
  - Safer - you can do what you like with the variable inside the function, and you won't affect what goes on outside the function
  - Inefficient - the computer has to make a copy of all variables - takes time and money
- Passing by reference
  - Efficient - no copying of data

Pros and cons

- Passing by value
  - Safer - you can do what you like with the variable inside the function, and you won't affect what goes on outside the function
  - Inefficient - the computer has to make a copy of all variables - takes time and money
- Passing by reference
  - Efficient - no copying of data
  - Dangerous - anything you do to the variables inside the function affects their value outside

Passing parameters - conclusion

- Most 3GLs let you choose which of the above is appropriate ot the circumstance

Passing parameters - conclusion

- Most 3GLs let you choose which of the above is appropriate ot the circumstance
- Some (e.g., C) allow you to specify whether variables passed by value can be changed within the function - best of both worlds!

Passing parameters - conclusion

- Most 3GLs let you choose which of the above is appropriate ot the circumstance
- Some (e.g., C) allow you to specify whether variables passed by value can be changed within the function - best of both worlds!
- Base R only supports passing by value

Passing parameters - conclusion

- Most 3GLs let you choose which of the above is appropriate ot the circumstance
- Some (e.g., C) allow you to specify whether variables passed by value can be changed within the function - best of both worlds!
- Base `R` only supports passing by value
- So how do you get information **out** of a function in `R`?

Passing parameters - conclusion

- Most 3GLs let you choose which of the above is appropriate ot the circumstance
- Some (e.g., C) allow you to specify whether variables passed by value can be changed within the function - best of both worlds!
- Base `R` only supports passing by value
- So how do you get information **out** of a function in `R`?
  - Use `return` - see next section!

Module interfaces - passing information out

- The main way to return values from functions is via the `return` statement

```
print.and.multiply <- function (x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
```

Module interfaces - passing information out

- The main way to return values from functions is via the `return` statement

```
print.and.multiply <- function (x, y) {
    cat('Inside function x=', x, 'y=', y, '\n')
    return(x * y)
}
```

- You can have more than one `return` statment (see factorial function earlier)

- If you want to return more than one variable, use a named list

```
print.and.multiply <- function (x, y) {
    return(list(x = x, y = y, mult = x * y))
}
print.and.multiply(10,20)
```

```
## $x
## [1] 10
##
## $y
## [1] 20
##
## $mult
## [1] 200
```

Best practice for modular programming

- Pass everything requried by the function in as an argument

Best practice for modular programming

- Pass everything requried by the function in as an argument
- Pass everything out using a `return` statement

Best practice for modular programming

- Pass everything requried by the function in as an argument
- Pass everything out using a `return` statement
- No global side effects

Best practice for modular programming

- Pass everything requried by the function in as an argument
- Pass everything out using a `return` statement
- No global side effects
- See next lecture for details, and many other tips for good programming practice. . .