Session 5
Control Flow 1

R for Social Data Science

Jeffrey Ziegler, PhD

Assistant Professor in Political Science & Data Science
Trinity College Dublin

Fall 2022

Road map for today

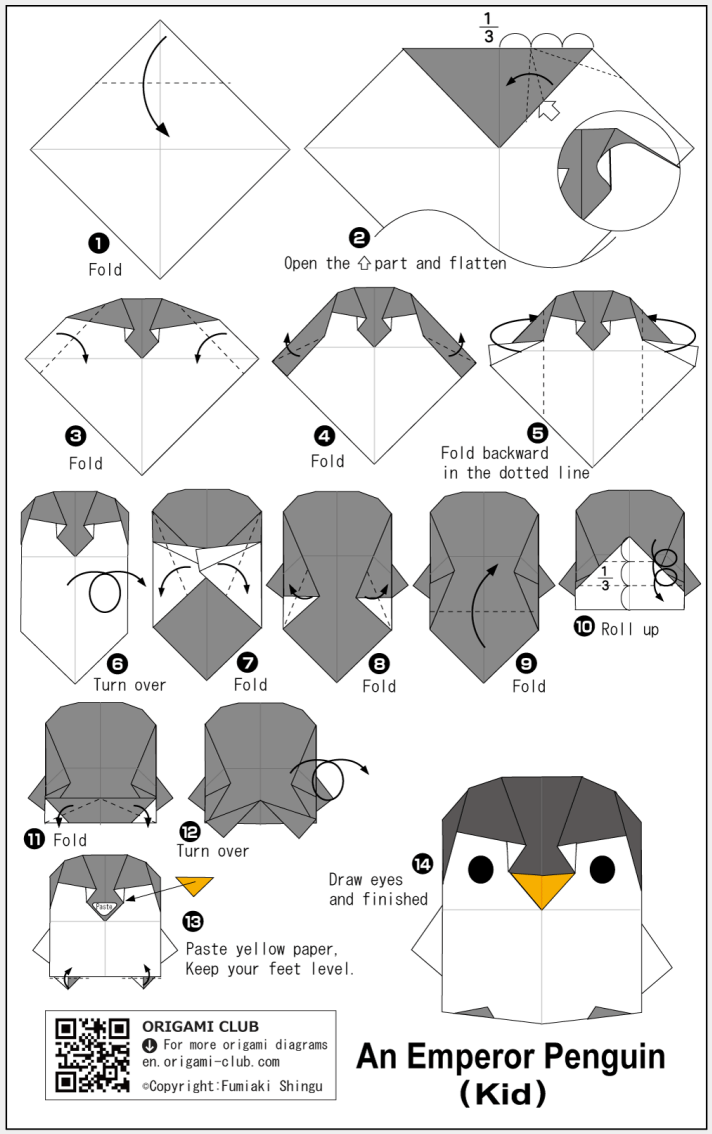
This week:

- Straight-line and branching programs
- Algorithms
- Conditional statements

Next time:

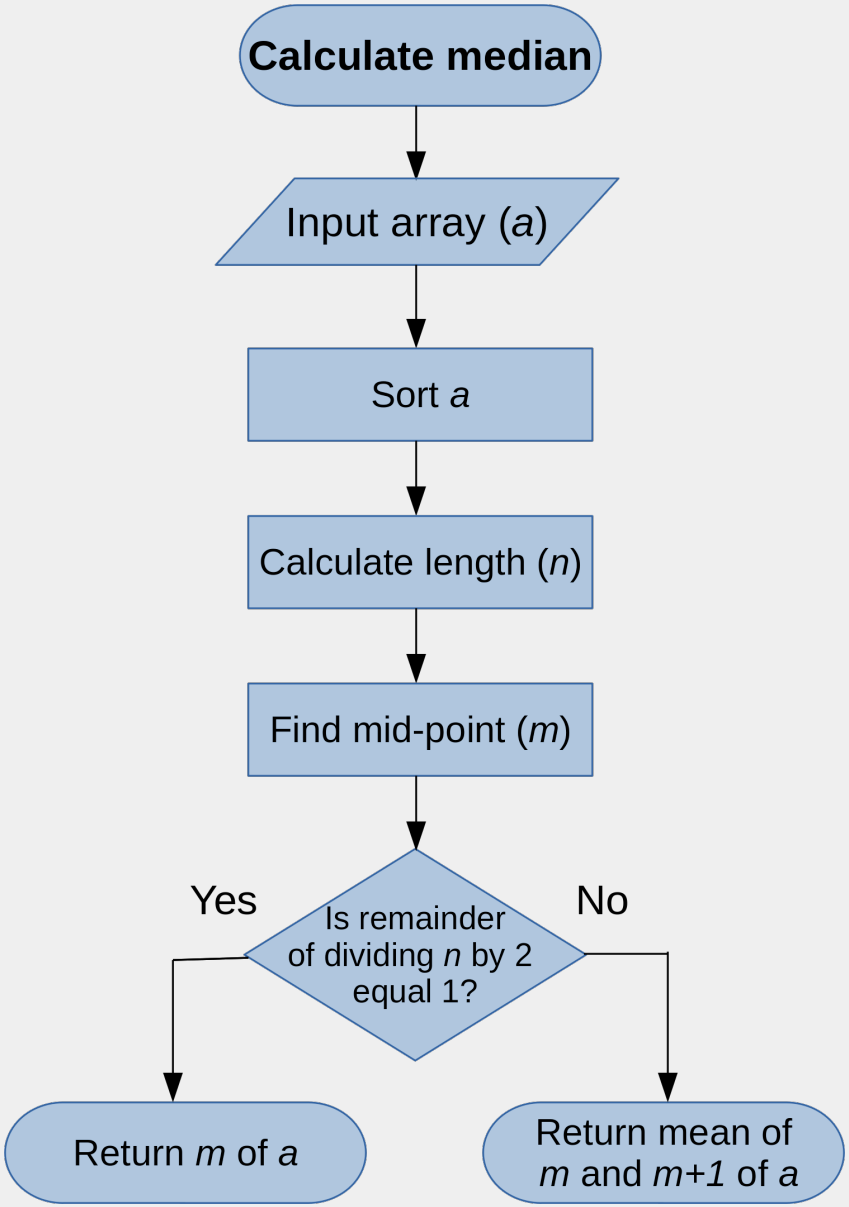
- Loops and Iteration

Algorithm Example

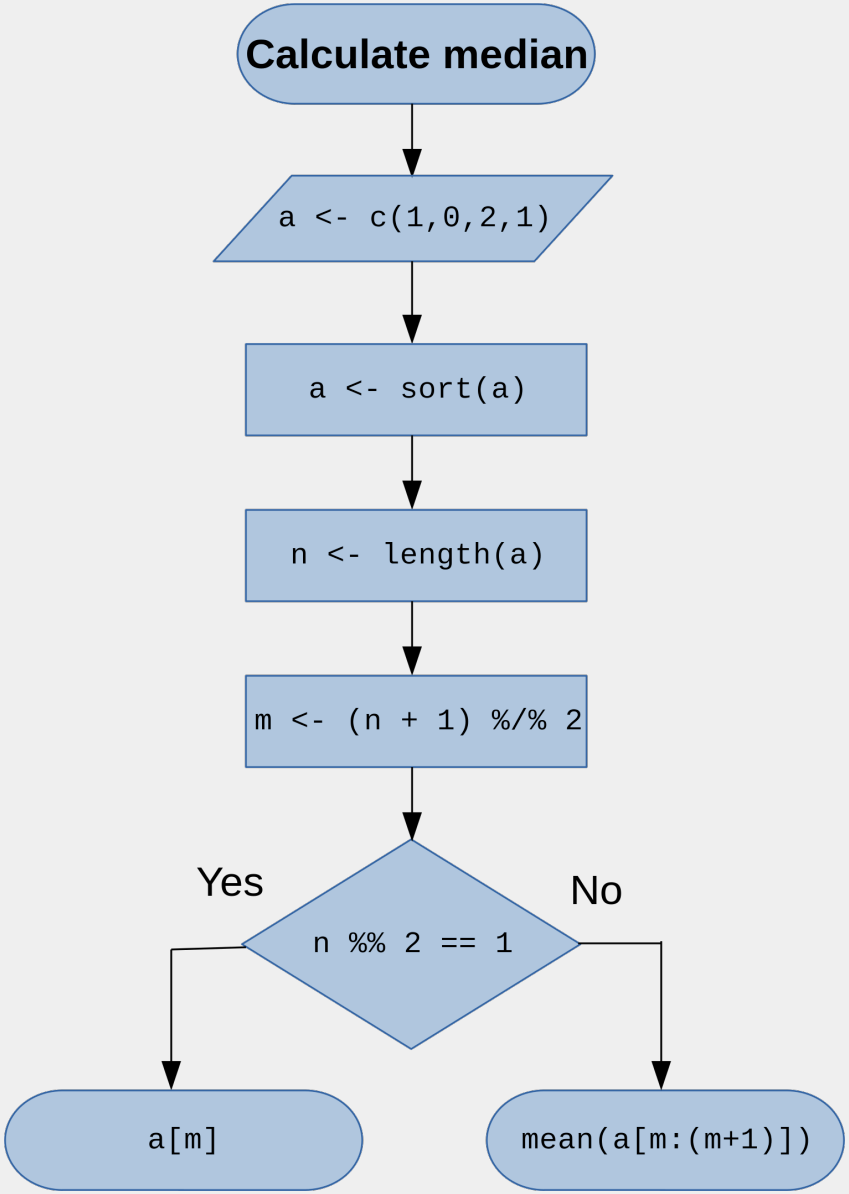


Source: Origami Club

Algorithm Example: "Straight line"



Algorithm flowchart (R)



Calculate median

```
a <- c(1,0,2,1) # Input vector (1-dimensional array)
a <- sort(a) # Sort vector
a
```

```
[1] 0 1 1 2
```

```
n <- length(a) # Calculate length of vector 'a'
n
```

```
[1] 4
```

```
m <- (n + 1) %/% 2 # Calculate mid-point, %/% is operator
    for integer division
m
```

```
[1] 2
```

Calculate median

```r
n %% 2 == 1 # Check whether the number of elements is odd,
   %% (modulo) gives remainder of division
```

```
[1] FALSE
```

```r
mean(a[m:(m+1)])
```

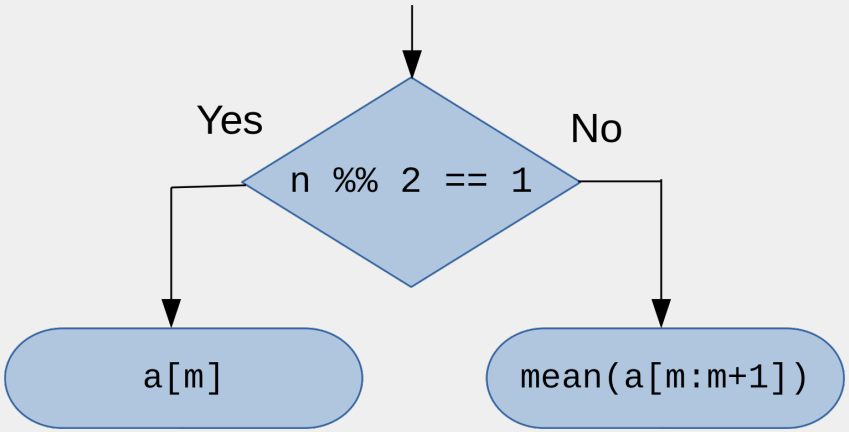
```
[1] 1
```

Control flow in R

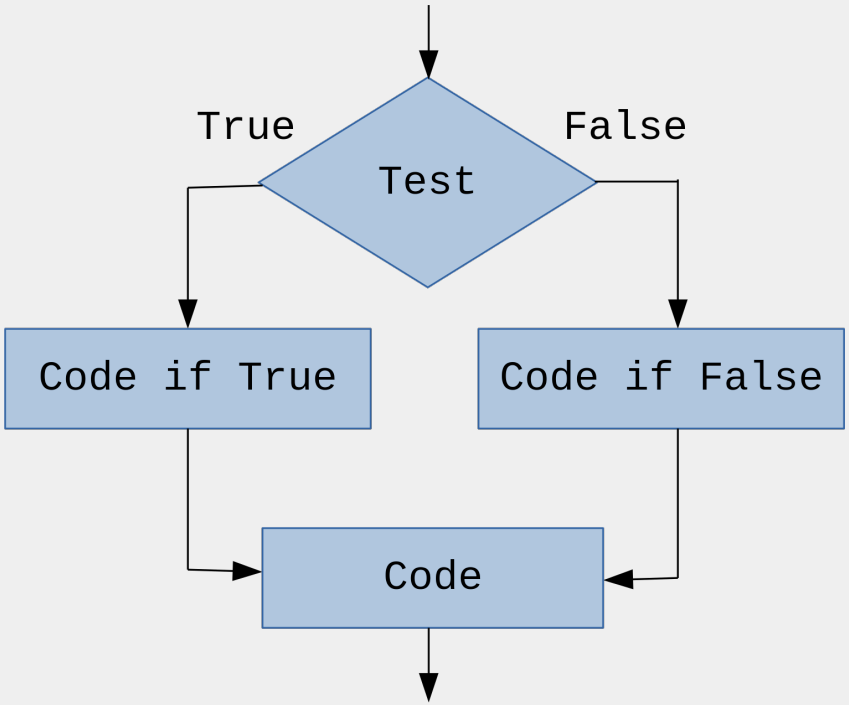
- *Control flow* is order statements are executed or evaluated
- Main ways of control flow in R:
  - *Branching* (conditional) statements (e.g. ‘if‘)
  - *Iteration* (*loops*) (e.g. ‘for‘)
  - *Function calls* (e.g. ‘length()‘)

Extra: R documentation on control flow

Branching programs



Conditional statements



Conditional statements: 'if'

- 'if' - defines condition under which some code is executed

```
if (<boolean_expression>) {
<some_code>
}
```

```
1 a <- c(1, 0, 2, 1, 100)
2 a <- sort(a)
3 n <- length(a)
4 m <- (n + 1) %/% 2
5 if(n %% 2 == 1){
6   a[m]
7 }
```

```
[1] 1
```

Conditional statements: 'if - else'

- 'if - else' - defines both condition under which some code is executed and alternative code to execute

```
if (<boolean_expression>) {
<some_code>
}else {
<some_other_code>
}
}
```

```
1 a <- c(1, 0, 2, 1)
2 a <- sort(a)
3 n <- length(a)
4 m <- (n + 1) %/% 2
5 if(n %% 2 == 1){
6   a[m]
7 }else{
8   mean(a[m:(m+1)])
9 }
```

```
[1] 1
```

Conditional statements: 'if - else if - else'

- 'if - else if - ... - else' - defines both condition under which some code is executed and several alternatives

```
if (<boolean_expression>){
<some_code>
} else if(<boolean_expression>) {
<some_other_code>
} else if(<boolean_expression>){
...
...
} else{
<some_more_code>
}
```

Example of longer conditional statement

```
x <- 42
if (x > 0){
  print("Positive")
}else if (x < 0){
  print("Negative")
}else{
  print("Zero")
}
```

```
[1] "Positive"
```

Optimising conditional statements

- Parts of conditional statement are evaluated sequentially, so makes sense to put most likely condition as first one

```r
# Ask for user input and assign as double
num <- as.double(readline("Please, enter a number:"))
if(num %% 2 == 0) {
  print("Even")
} else if(num %% 2 == 1) {
  print("Odd")
} else{
  print("This is a real number")
}
```

```
Please, enter a number: 43
[1] "Odd"
```

Nesting conditional statements

- Conditional statements can be nested within each other
- But consider code legibility, modularity, and speed

```r
num <- as.integer(readline("Please, enter a number:"))
# Ask for user input and cast as integer
if (num > 0) {
  if (num %% 2 == 0){
  print("Positive even")
  }
  } else{
  print("Positive odd")
}else if (num < 0){
  if (num %% 2 == 0) {
  print("Negative even")
  # Notice that odd/even check appears twice
  } else{
  print("Negative odd")
  # Consider abstracting this as a function
  }
}else{
  print("Zero")
}
```

```
Please, enter a number: -43
[1] "Negative odd"
```

'ifelse()' function

- R also provides a vectorized version of 'if - else' construct
- It takes a vector as an input and returns another vector as an output

```
ifelse(<boolean_expression>, <if_true>, <if_false>)
```

```
num <- 1:10
num
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
ifelse(num %% 2 == 0, "even", "odd")
```

```
 [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"
"even" "odd"  "even"
```

Note on code formatting

- Use consistent style and indentation (RStudio indents by 2 whitespaces)
- Even though it doesn't affect how programs are executed

```r
# Good style
is_positive <- function(num){
  if(num>0){
    res <- TRUE
  }
  else{
    res <- FALSE
  }
  return(res)
}
```

```r
# Bad style
is_positive <- function(num){
if(num>0){
res <- TRUE
}
else{
res <-FALSE
}
return(res)
}
```

Exercise: Conditional Statements

- Below you will find a code snippet for finding the maximum value in vector ‘v‘ using exhaustive enumeration
- Modify it in such a way that it finds the minimum (rather than maximum) value
- Check that your code works correctly by applying the built-in function ‘min()‘

```r
# Below you will find a code snippet for finding the maximum value in vector ‘v‘
    using exhaustive enumeration
# Modify it in such a way that it finds the minimum (rather than maximum) value
# Check that your code works correctly by applying the built-in function ‘min()
set.seed(2022)
v <- sample(1:1000,50)
max_val <- v[1]
for(i in v){
  if(i > max_val){
    max_val <- i
  }
}
max_val
```

```
[1] 998
```

Exercise: Conditional Statements

- Now let's make this code more robust
- Re-write the code above so that it can handle vectors that contain NAs in them
- Test your code to find min() on vector below

```
set.seed(2022)
v <- sample(c(1:500,rep(NA,500)),25)
```

Exercise: Conditional Statements

```r
min_val <- v[1]
for(i in v[!is.na(v)]){
  if(i < min_val){
    min_val <- i
  }
}
min_val
```

```
[1] 7
```

Wrap up

This week:

- Straight-line and branching programs
- Algorithms
- Conditional statements

Next time:

- Loops and Iteration