Basics

First Step

Installing course package before start

```
#install.packages("ISwR")
#install.packages("ggplot2")
library(ISwR)
library(ggplot2)
```

Vectorized arithmetic

One strength of R is that it can handle entire *data vectos* as single objects.
The operation is carried out elementwise for the code shown.

```
weight <- c(60,72,57,90,95,72)
height <- c(1.75,1.80,1.65,1.90,1.74,1.91)
bmi <- weight/height^2
bmi
```

```
## [1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
```

It is in fact possible to perform arithmetic operations on vectors of different length. We already used that when we calculated the `height^2` part above since `2` has length 1. In such cases, the shorter vector is ***recycled***.
A warning is issued if the longer vector is not a multiple of the shorter in length.
These conventions for vectorized calculations make it very easy to specify typical statistical calculations. Consider, for instance, the calculation of the mean and standard deviation of the `weight` variable.
mean, $\bar{x} = \sum \frac{x_i}{n}$:

```
sum(weight)
```

```
## [1] 446
```

```
xbar <- sum(weight)/length(weight)
xbar
```

```
## [1] 74.33333
```

standard deviation, $s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$

```r
sd <- sqrt(sum((weight-xbar)^2)/(length(weight)-1))
sd
```

```
## [1] 15.42293
```

Since R is a statistical program, such calculations are already built into the program, and you get the same results just by entering

```r
mean(weight)
```

```
## [1] 74.33333
```

```r
sd(weight)
```

```
## [1] 15.42293
```

Standard Procedures

The BMI for a normal-weight individual should be between 20 and 25, and we want to know if our data deviate systemically from that. We can use a one-sample t test to assess whether the six persons' BMI can be assumed ot have mean 22.5 given that they come from a normal distribution.

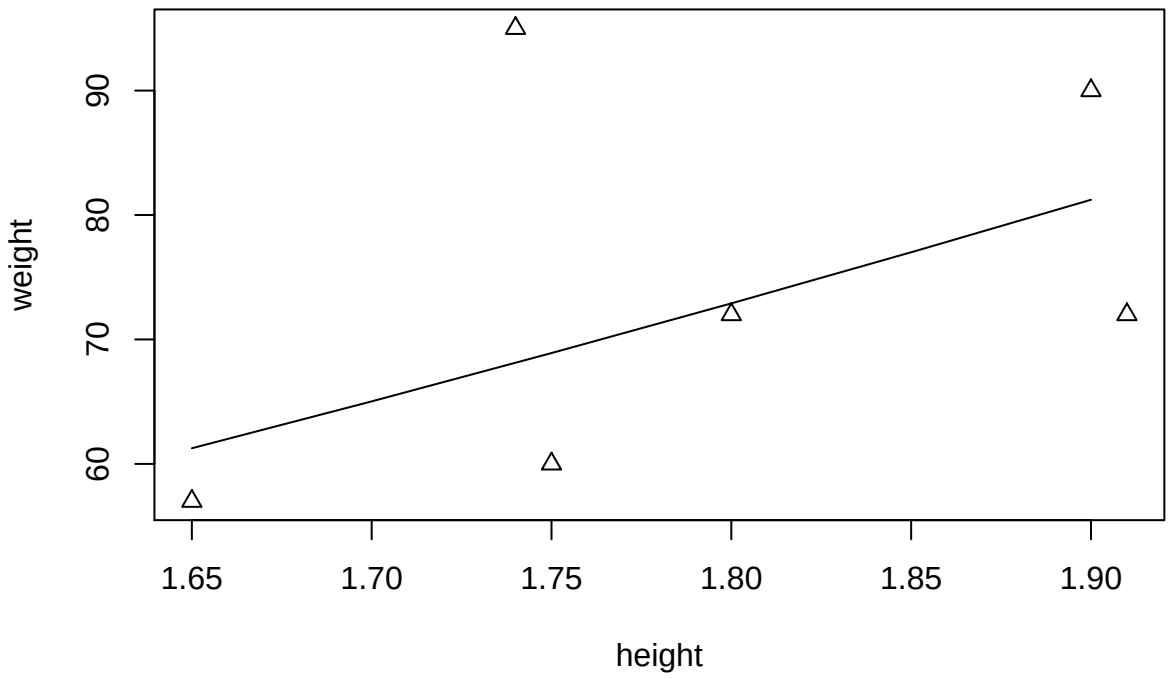
```r
t.test(bmi, mu = 22.5)
```

```
##
##  One Sample t-test
##
## data:  bmi
## t = 0.34488, df = 5, p-value = 0.7442
## alternative hypothesis: true mean is not equal to 22.5
## 95 percent confidence interval:
##  18.41734 27.84791
## sample estimates:
## mean of x
##  23.13262
```

Graphics

The idea behind the BMI calculation is that this value should be independent of the person's height, thus giving us a single number as an indication of whether someone is overweight and by how much. Since a normal BMI should be about 22.5, we would expect that $weight \approx 22.5 \times height^2$. Accordingly, we can superimpose a curve of expected weights at BMI 22.5 on the figure.

```r
plot(height,weight, pch=2) #pch - plotting character
hh <- c(1.65,1.70,1.75,1.80,1.85,1.90)
weight.BMI22.5 <- 22.5*hh^2
lines(hh, weight.BMI22.5)
```

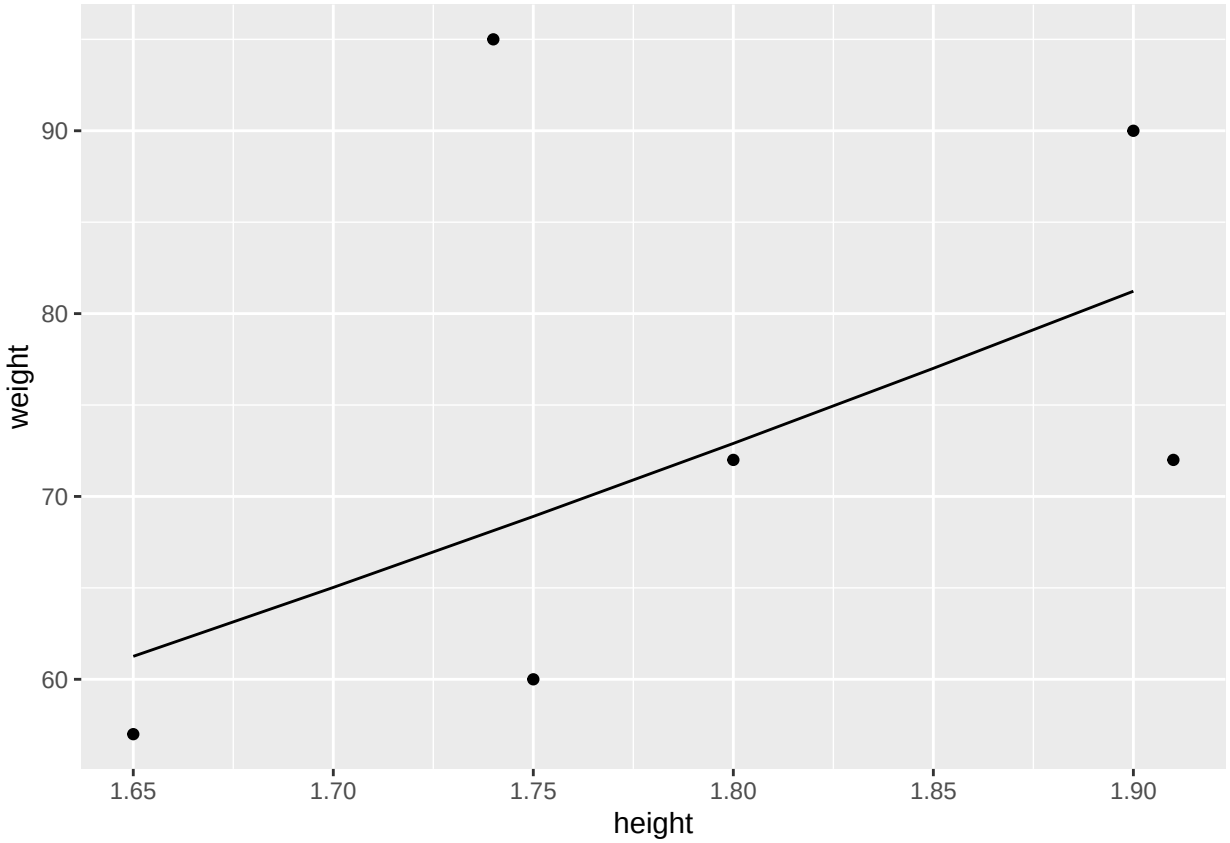


ggplot2 equivalent:

```
ggplot(as.data.frame(height,weight),mapping=aes(x=height,y=weight)) +
  geom_point()+
  geom_line(data=as.data.frame(hh,weight.BMI22.5), mapping = aes(hh,weight.BMI22.5))
```

```
## Warning in as.data.frame.numeric(height, weight): 'row.names' is not a character
## vector of length 6 -- omitting it. Will be an error!
```

```
## Warning in as.data.frame.numeric(hh, weight.BMI22.5): 'row.names' is not a
## character vector of length 6 -- omitting it. Will be an error!
```



Vectors

Numeric vector

```
c(1,2,3)
```

```
## [1] 1 2 3
```

Character vector

```
c("Huey","Dewey","Louie")
```

```
## [1] "Huey"  "Dewey" "Louie"
```

Logical vector

```
c(TRUE,TRUE,FALSE,TRUE)
```

```
## [1]  TRUE  TRUE FALSE  TRUE
```

Functions that create vectors

`c()`, concatenation

```
x<- c(1,2,3)
y <- c(10,20)
z <- c(x,y,100)
z
```

```
## [1]   1   2   3  10  20 100
```

```
# Assigning names to the elements
x <- c(red="Huey", blue="Dewey", green="Louie")
x
```

```
##     red    blue   green
##  "Huey" "Dewey" "Louie"
```

```
names(x) # To extract the names
```

```
## [1] "red"   "blue"  "green"
```

`seq()`,sequence

```
seq(4,10,2)
```

```
## [1]  4  6  8 10
```

```
4:9 # For step size equals to 1
```

```
## [1] 4 5 6 7 8 9
```

`rep()`, replicate Generate repeated values, depending on whether the second argument is a vector or a single number

```
oops <- c(7,9,13)
rep(oops,3) # Repeat oops thrice
```

```
## [1]  7  9 13  7  9 13  7  9 13
```

```
rep(oops,1:3) # Repeat first element once, second element twice, third element thrice
```

```
## [1]  7  9  9 13 13 13
```

The 'rep function is often used for things such as group codes: IF it is known that the first 10 observations are men and the last 15 ar women, we can use

```
rep(1:2,c(10,15))
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

The special case where there are equally many replications of each value can be obtained using the `each` argument.

```
rep(1:2, each=10) # same as rep(1:2, c(10,10))
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
```

Quoting and escape sequences

If we print a character vector, it usually comes out with quotes added to each element. There is a way to avoid this, namely to use the `cat` function.

```
cat(c("Huey","Dewey","Louie"))
```

```
## Huey Dewey Louie
```

This prints the strings without quotes, just separated by a space character. There is no newline following the string, so the prompt (`>`) for the enxt line of input follows directly at the end of the line.
Notice that when the character vector is printed by `cat`, there is no way of telling the difference from the single string `"Huey Dewey Louie"`.
To get the system ptompr onto the next line, we must include a newline character

```
cat("Huey","Dewey","Louie","\n")
```

```
## Huey Dewey Louie
```

The backslash (\) is know as the *escape character*

```
cat("What is \"R\"?\n")
```

```
## What is "R"?
```

Matrices and Arrays

```
x <- 1:12
dim(x) <- c(3,4)
x
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

The `dim` assignment function sets or changes the *dimension attribute* of `x`, causing R to treat the vector of 12 numbers as a 3 x 4 matrix.

```
matrix(1:12, nrow=3, byrow=TRUE)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

Notice how the `byrow = TRUE` switch causes the matrix to be filled in a rowwise fashion rather than columnwise.
Useful functions that operate on matrices include `rownames`, `colnames` and the transposition function `t`

```
x <- matrix(1:12, nrow=3, byrow = TRUE)
rownames(x) <- LETTERS[1:3]
x
```

```
##   [,1] [,2] [,3] [,4]
## A    1    2    3    4
## B    5    6    7    8
## C    9   10   11   12
```

```
t(x)
```

```
##      A B  C
## [1,] 1 5  9
## [2,] 2 6 10
## [3,] 3 7 11
## [4,] 4 8 12
```

The character vector `LETTERS` is a built-in variable that contains the capital letters A-Z. Similar useful vectors are `letters`, `month.name`, and `month.abb`
We can glue vectors together using `rbind()` or `cbind()`

```
cbind(A=1:4,B=5:8,C=9:12)
```

```
##      A B  C
## [1,] 1 5  9
## [2,] 2 6 10
## [3,] 3 7 11
## [4,] 4 8 12
```

```
rbind(A=1:4,B=5:8,C=9:12)
```

```
##   [,1] [,2] [,3] [,4]
## A    1    2    3    4
## B    5    6    7    8
## C    9   10   11   12
```

Factors

It is common in statistical data to havecategorical variables, indicating some subdivision of data, and typically these are input using a numeric code.
There are analyses where it is essential for R to be able to distinguish between categorical codes and variables whose values have a direct numerical meaning.
The terminology is that a factor has a set of ***levels***. Internally, a n-level factor consists of two items: (a) a vector of integers between 1 and n, and (b) a character vector of length n containing strings describing what the n levels are.

```
pain <- c(0,3,2,2,1)
fpain <- factor(pain, levels = 0:3) # treat the numeric vector as categorical variable
levels(fpain) <- c("none","mild","medium","severe")
fpain
```

```
## [1] none   severe medium medium mild
## Levels: none mild medium severe
```

```
as.numeric(fpain)
```

```
## [1] 1 4 3 3 2
```

```
levels(fpain)
```

```
## [1] "none"   "mild"   "medium" "severe"
```

R also allows us to create a special kind of factor in which the levels are ordered using the `ordered` function, which works similarly to `factor`. These are potentially useful in that they distinguisg nominal and ordinal variables from each other.

Lists

`List` is useful to combine a collection of objects of **different types** into a larger composite object.
As an example, consider a set of data from Altman (1991, p.183) concerning pre- and postmenstrual energy intak ein a group of women.

```
intake.pre <- c(5260,5470,5640,6180,6390,
+ 6515,6805,7515,7515,8230,8770)
intake.post <- c(3910,4220,3885,5160,5645,
+ 4680,5265,5975,6790,6900,7335)
myList <- list(before=intake.pre, after=intake.post)
myList
```

```
## $before
##  [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
##
## $after
##  [1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900 7335
```

The componenets of the list are named according to the argument names used in `list`. Named components may be extracted as follows:

```
myList$before
```

```
##  [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
```

Data Frames

A data frame is a list of vectors and/or factors of the same length that are related "across" such that data in the same position come from the same experimental unit. In addition, it has a unique set of row names.

```
d <- data.frame(intake.pre, intake.post)
d
```

```
##    intake.pre intake.post
## 1        5260        3910
## 2        5470        4220
## 3        5640        3885
## 4        6180        5160
## 5        6390        5645
## 6        6515        4680
## 7        6805        5265
## 8        7515        5975
## 9        7515        6790
## 10       8230        6900
## 11       8770        7335
```

Note that these data are paired, that is the same women has an intak eof 5260kJ premenstrually and 3910kJ postmenstrually.
As with lists, components can be accessed using the $ notation.

```
d$intake.pre
```

```
##  [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
```

Indexing

If we need the premenstrual energy intake for woman no.5, we can

```
intake.pre[5]
```

```
## [1] 6390
```

We can reassign the values of the vectors/list/matrices/data frames using indexing method

```
intake.pre[5] <- 6600
intake.pre[5]
```

```
## [1] 6600
```

To subset the data, for instance the 3rd, 5th and 7th data,

```
intake.pre[c(3,5,7)] #intake,pre[3,5,7] would mean indexing into a 3D array
```

```
## [1] 5640 6600 6805
```

```
intake.pre[1:5]
```

```
## [1] 5260 5470 5640 6180 6600
```

We can use negative indexing to get all observations **except** those in the negative indexes.

```
intake.pre[-c(3,5,7)] # Everything except 3rd, 5th and 7th
```

```
## [1] 5260 5470 6180 6515 7515 7515 8230 8770
```

It is not possible to mix positive and negative indices. That would be highly ambiguous.

Conditional selection

To subset the postmenstrual energy intake for the fouor women who had an energy intake above 7000kJ premenstrually.

```
intake.post[intake.pre>7000]
```

```
## [1] 5975 6790 6900 7335
```

This kind of expression makes sense only if the variables that go into the relational expression have the same length as the variable being indexed.
To combine several expressions, we can use the logical operators `&` (logical "and"), `|` (logical "or"), and `!` (logical "not").

```
intake.post[intake.pre>7000 & intake.pre<=8000]
```

```
## [1] 5975 6790
```

There are also `&&` and `||`, which are used for flow control in R programming. They will not be discussed here.
The result of the logical expression is a logical vector.

```
intake.pre > 7000 & intake.pre <= 8000
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

Indexing with a logical vector implies that we pick out the values where the logical vector is `TRUE`. If missing values `NA` appear in an indexing vector, then R will create the corresponding elements in the result but set the values to `NA`.
In addition to the relational and logical operators, there are a series of functions that return a logical value. A particularly important one is `is.na(x)`, which is used to find out which elements of `x` are recorded as missing, `NA`.
Notice that there is a real need for `is.na` because we cannot make comparisons of the form `x==NA`. That simply gives `NA` as the result for any value of `x`. The result of a comparison with an unknown value is unknown.

Indexing of data frames

```
d <- data.frame(intake.pre, intake.post)
d[5,1] # fifth row first column: 5th sample for "pre" measurement
```

```
## [1] 6600
```

```
d[5,] # all data for 5th sample
```

```
##   intake.pre intake.post
## 5       6600        5645
```

```
d[2] # equivalent to d[,2]
```

```
##    intake.post
## 1         3910
## 2         4220
## 3         3885
## 4         5160
## 5         5645
## 6         4680
## 7         5265
## 8         5975
## 9         6790
## 10        6900
## 11        7335
```

```
d[d$intake.pre>7000,] # rows of data where intake.pre>7000
```

```
##    intake.pre intake.post
## 8        7515        5975
## 9        7515        6790
## 10       8230        6900
## 11       8770        7335
```

```
d[1:2,] # first 2 sample
```

```
##   intake.pre intake.post
## 1       5260        3910
## 2       5470        4220
```

```
head(d) # first 6 line, use head(d,n) for frist n line
```

```
##   intake.pre intake.post
## 1       5260        3910
## 2       5470        4220
## 3       5640        3885
## 4       6180        5160
## 5       6600        5645
## 6       6515        4680
```

```
tail(d) # ending 6 lines
```

```
##    intake.pre intake.post
## 6        6515        4680
## 7        6805        5265
## 8        7515        5975
## 9        7515        6790
## 10       8230        6900
## 11       8770        7335
```

Grouped data and data frames

The natural way of storing grouped data in a data frame is to have the data themselves sin one vector and parallel to that have a factor telling which data are from which group. Consider, for instance, the following data set on energy expenditure for lean and obese women.

```
energy
```

```
##    expend stature
## 1    9.21   obese
## 2    7.53    lean
## 3    7.48    lean
## 4    8.08    lean
## 5    8.09    lean
## 6   10.15    lean
## 7    8.40    lean
## 8   10.88    lean
## 9    6.13    lean
## 10   7.90    lean
## 11  11.51   obese
## 12  12.79   obese
## 13   7.05    lean
## 14  11.85   obese
## 15   9.97   obese
## 16   7.48    lean
## 17   8.79   obese
## 18   9.69   obese
## 19   9.68   obese
## 20   7.58    lean
## 21   9.19   obese
## 22   8.11    lean
```

This is a convenient format since it generalizes easily to data classified by multiple criteria. However, sometimes it is desirable to have data in a separate vector for each group..

```
exp.lean <- energy$expend[energy$stature=="lean"]
exp.obese <- energy$expend[energy$stature=="obese"]
exp.lean
```

```
##  [1]  7.53  7.48  8.08  8.09 10.15  8.40 10.88  6.13  7.90  7.05  7.48  7.58
## [13]  8.11
```

```
exp.obese
```

```
## [1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69  9.68  9.19
```

Alternatively, we can use the `split` function, which generates a list of vectors according to a grouping

```
# split(x,f), split data in x into the groups defined by f
l <- split(energy$expend, energy$stature)
l
```

```
## $lean
##  [1]  7.53  7.48  8.08  8.09 10.15  8.40 10.88  6.13  7.90  7.05  7.48  7.58
## [13]  8.11
##
## $obese
## [1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69  9.68  9.19
```

Implicit loops

A common application of loops is to apply a function to each element of a set of values or vectors and collect the results in a single structure. In R this is abstracted by the functions `lapply` and `sapply`. The former always returns a list, whereas the latter tries to simplify the result to a vector or a matrix if possible.
So, to compute the mean of each variable in a data frame of numeric vectors, we can do

```
head(thuesen)
```

```
##   blood.glucose short.velocity
## 1          15.3           1.76
## 2          10.8           1.34
## 3           8.1           1.27
## 4          19.5           1.47
## 5           7.2           1.27
## 6           5.3           1.49
```

```
lapply(thuesen, mean, na.rm=TRUE)
```

```
## $blood.glucose
## [1] 10.3
##
## $short.velocity
## [1] 1.325652
```

```
sapply(thuesen, mean, na.rm=TRUE)
```

```
##  blood.glucose short.velocity
##      10.300000       1.325652
```

Sometimes we want to repeat something a number of times but still collect the results as a vector. This makes sense only when the repeated computations actually give different results, the common case being simulation studies. This can be done using `sapply`, but there is a simplified version called `replicate`, in which we just have to give a count and the expression to evaluate.

```
replicate(10, mean(rexp(20)))
```

```
##  [1] 1.2214967 1.2040218 1.0164812 0.8205691 1.3556495 0.7656640 1.0664656
##  [8] 1.1521562 0.9018211 0.8081153
```

A similar function, `apply`, allows us to apply a function to the rows or columns of a matrix, as in

```
m <- matrix(rnorm(12),nrow = 4)
m
```

```
##            [,1]        [,2]        [,3]
## [1,] -0.9906507  0.90617283 -0.14388885
## [2,] -0.5849673 -1.03657826  0.02405306
## [3,] -1.3887359  0.82911174 -0.06664292
## [4,] -0.6970527  0.01652476  0.79218939
```

```
apply(m,2,min)
```

```
## [1] -1.3887359 -1.0365783 -0.1438889
```

The second argument is the index (or vector of indices) that defines whether the function is applied columnwise, rowwise or both.
The function `tapply` allows us to create tables of the value of a function on subgroups/ factors defined by its second argument, which can be a factor or a list of factors. In the latter case a cross-classified table is generated.

```
tapply(energy$expend,energy$stature, median)
```

```
##  lean obese
##  7.90  9.69
```

Sorting

```
intake
```

```
##     pre post
## 1  5260 3910
## 2  5470 4220
## 3  5640 3885
## 4  6180 5160
## 5  6390 5645
## 6  6515 4680
## 7  6805 5265
## 8  7515 5975
## 9  7515 6790
## 10 8230 6900
## 11 8770 7335
```

```
intake$post
```

```
##  [1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900 7335
```

```
sort(intake$post)
```

```
##  [1] 3885 3910 4220 4680 5160 5265 5645 5975 6790 6900 7335
```

Sorting a single vector is not always what is required. Often we need to sort a series of variables according to the values of some other variables - blood pressures sorted by sex and age, for instance.
For this purpose, we first compute an *ordering* of a variable.

```
order(intake$post)
```

```
##  [1]  3  1  2  6  4  7  5  8  9 10 11
```

Interpreting the result of `order` is a bit tricky - it should be read as follows: We sorted `intake$post` by placing its values in the order no.3, no.1, no.2, no.6, etc.
The point is that, by indexing with this vector, other variables can be sorted by the same criterion. Note that indexing with a vector containing the numbers from 1 to the number of elements exactly once corresponds to a reordering of the elements.

```
o <- order(intake$post)
intake$post[o]
```

```
##  [1] 3885 3910 4220 4680 5160 5265 5645 5975 6790 6900 7335
```

```
intake$pre[o]
```

```
##  [1] 5640 5260 5470 6515 6180 6805 6390 7515 7515 8230 8770
```

It is of course also possible to sort the entire data frame `intake`

```
intake.sorted <- intake[o,]
intake.sorted
```

```
##     pre post
## 3  5640 3885
## 1  5260 3910
## 2  5470 4220
## 6  6515 4680
## 4  6180 5160
## 7  6805 5265
## 5  6390 5645
## 8  7515 5975
## 9  7515 6790
## 10 8230 6900
## 11 8770 7335
```

Sorting by several criteria is done simply by having several arguments to `order`; for instance, `order(sedx,age)` will give a main division into men and women, and within each sex an ordering by age. The second variable is used when the order cannot be decided from the first variable. Sorting in reverse order can be handled by, for example, changing the sign of the variable.