A Tutorial: Some Fundamentals of R

R is an object-oriented language. For us, the most fundamental objects are vectors. To begin, you can think of a *vector* as an ordered list of numbers called *elements*. The *index* of an element specifies its position in the list: index 1 for the first element, index 2 for the second, and so on.

Defining a vector.

We begin by showing a few ways to specify the components of a vector. You should follow through this tutorial seated at a computer running R. The operator `<-` is used to assign values. For example, `n <- 25` means that `n` is a vector with 25 as its only element. Below we show how this looks in the R Console window. (There is no "print" statement in R; to print an object on the screen just type its name.) Can you duplicate this printout?

```
> n <- 25
> n
[1] 25
```

Names of objects in R are case sensitive. For example `n` and `N` are different objects. If you have not previously defined `N`, then typing `N` at the `>` prompt in the Console window will give the following.

```
> N
Error: Object "N" not found
```

The symbol `c` can be used to make a vector with several elements. You can think of `c` as standing for "combine." But R ordinarily treats vectors as columns of entries (even if they are sometimes printed out in rows), so you may prefer to think as `c` as standing for "column."

```
> v1 <- c(7, 2, 3, 5)
> v1
[1] 7 2 3 5
```

Here are a few convenient shorthand notations for making vectors. They are often used to make vectors that are too long to make using the `c`-notation and that have elements following a particular pattern.

```
> v2 <- numeric(4)
> v2
[1] 0 0 0 0

> numeric(10)
 [1] 0 0 0 0 0 0 0 0 0 0
```

What do you suppose `numeric(100)` does? Try it. Can you figure out the meaning of the bracketed numbers at the beginning of each printed line?

This method is often used to "initialize" a very long vector of 0s that is to be used in a loop. Inside the loop, each element may be changed, one at a time, from 0 to some other value. More about loops later.

The next notation allows you to make a vector where all elements are equal to a specified value (not necessarily 0).

```
> v3 <- rep(3, 4)
> v3
[1] 3 3 3 3

> rep(4, 3)
[1] 4 4 4
```

Verify that there is no difference between the effect of `numeric(4)` and of `rep(0, 4)`.

The colon notation is used for consecutive integers.

```
> v4 <- 1:4
> v4
[1] 1 2 3 4
```

Predict the output of `4:-1`, and verify.

A sequence of equally spaced *noninteger values* can be specified as follows.

```
> seq(1, 2.2, by=.1)
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1
[13] 2.2
```

In the `seq` notation above there are three arguments (in order of appearance): the *beginning* number, the *ending* number, and the *increment* between successive elements of the vector. This vector has 13 elements.

Alternatively, the third argument can be used specify the length of the vector of equally spaced values. The following code gives a vector with 11 elements.

```
> seq(1, 2.2, length=11)
 [1] 1.00 1.12 1.24 1.36 1.48 1.60 1.72 1.84 1.96 2.08
[11] 2.20
```

The notations `seq(1, 10)`, with no third argument, and `1:10` give the same result. Try it. What vector results from `seq(1, 10.5)`?

The `c`-symbol can be used to combine vectors as well as single numbers.

```
> v5 <- c(v3, v4, 7)
> v5
[1] 3 3 3 3 1 2 3 4 7
```

Simple arithmetic.

Operations on vectors are performed elementwise. In particular, when an operation involves *a vector and a number*, the number is used to modify each element of the vector as specified by the operation. Here are some examples.

```
> w1 <- 3*v1
> w1
[1] 21  6  9 15

> w2 <- v1/2
> w2
[1] 3.5 1.0 1.5 2.5

> w3 <- 5 - v1
> w3
[1] -2  3  2  0

> w4 <- w3^2
> w4
[1] 4 9 4 0
```

You should experiment with minor variations of them. Also, show that `(0:10)/10` gives the same result as `seq(0, 1, by=.1)`.

When arithmetic involves *two vectors of the same length*, then the operation applies to elements with the same index.

```
> w5 <- w3 + w4
> w5
[1]  2 12  6  0

> (5:1)*(1:5)
[1] 5 8 9 8 5

> (5:0)^(0:5)
[1] 1 4 9 8 1 0
```

Predict and verify the results of `w3 - w4`, `w4 - w3`, `w4 * w3`, `w4^w3`, and `w4/w3`. Here are a few more examples.

```
> (1:4)^2
[1]  1  4  9 16

> (5:1)/(1:5)
[1] 5.0 2.0 1.0 0.5 0.2
```

What do you suppose is the result of `(1:4)^(0:3)`? Verify your answer.

Indexes and Assignments.

Sometimes we want to deal with only one element of a vector. The index notation `[ ]` helps to do this. The simplest version of referencing by index is just to specify the index (position number within the vector) you want.

```
> w1
[1] 21  6  9 15
> w1[3]
[1] 9

> v5
[1] 3 3 3 3 1 2 3 4 7
> v5[9]
[1] 7
```

The bracket notation can be used along with the assignment operator `<-` to change the value of an element of an existing vector.

```
> v2
[1] 0 0 0 0

> v2[1] <- 6
> v2
[1] 6 0 0 0
```

The bracket notation can also be used to specify or change *more than one element* of a vector.

```
> v3 <- numeric(10)
> v3
 [1] 0 0 0 0 0 0 0 0 0 0

> v3[1:3] <- 4:6
> v3
 [1] 4 5 6 0 0 0 0 0 0 0

> v3[2:4]
[1] 5 6 0
```

We will see still other uses for the bracket notation later.

Operations with vectors of unequal length.

When the vectors operated upon are of unequal lengths, then the shorter vector is "recycled" as often as necessary to match the length of the longer vector. If the vectors are of different lengths because of a programming error, this can lead to unexpected results. But sometimes recycling of short vectors is the basis of clever programming. Also, all of the examples we have shown so far with operations involving a vector and a single number recycle the single number (which R regards as a one-element vector).

We illustrate with three examples of operations with vectors of unequal lengths. You should verify each of them by hand computation.

```
> (1:10)/(1:2)
 [1] 1 1 3 2 5 3 7 4 9 5

> (1:10)/(1:5)
 [1] 1.000000 1.000000 1.000000 1.000000 1.000000 6.000000
 [7] 3.500000 2.666667 2.250000 2.000000
```

Notice that R gives a warning message if the recycling comes out "uneven"; this situation often arises because of a programming error.

```
> (1:10)/(1:3)
 [1]  1.0  1.0  1.0  4.0  2.5  2.0  7.0  4.0  3.0 10.0
Warning message: longer object length
        is not a multiple of shorter object length
        in: (1:10)/(1:3)
```

Output formats.

In case you haven't figured it out by now, the bracketed number at the beginning of each line of printed output gives the index of the *first element* printed on the line. In the output for `(1:10)/(1:5)`, the 9th element is 9/4 = 2.25.

Depending on the width you have chosen for your Console window and the number of digits in each result, R may put various numbers of results on a line. With a narrower window, the display for `(1:10)/(1:5)` might have been printed as shown below.

```
> (1:10)/(1:5)
 [1] 1.000000 1.000000 1.000000 1.000000
 [5] 1.000000 6.000000 3.500000 2.666667
 [9] 2.250000 2.000000
```

There are other more complex ways to control the appearance of printed output, but this is enough for now.

Vector functions.

We begin with vector functions that return a *single number*. The meanings of the functions in the following demonstration should be self-explanatory.

```
> w2
[1] 3.5 1.0 1.5 2.5
> max(w2)
[1] 3.5

> w3
[1] -2  3  2  0
> mean(w3)
[1] 0.75

> v1
[1] 7 2 3 5
> sum(v1)
[1] 17

> v4
[1] 1 2 3 4
> prod(v4)
[1] 24

> v5
[1] 3 3 3 3 1 2 3 4 7
> length(v5)
[1] 9
```

By using parentheses for grouping, one can combine several expressions that involve functions.

```
> (sum(w3^2) - (sum(w3)^2)/length(w3)) / (length(w3) - 1)
[1] 4.916667
```

A simpler way to get the same result would be to use the `var` function.

```
> var(w3)
[1] 4.916667
```

Verify that the standard deviation of the numbers –1, 3, 2, 0 is computed by using the expression `sqrt(var(w3))`.

Some vector functions return *vectors*. For example, `sqrt` takes the square root of each element of a vector, and `cumsum` forms cumulative sums of the elements of a vector.

```
> sqrt(c(1, 4, 9, 16, 25))
[1] 1 2 3 4 5

> cumsum(1:5)
[1]  1  3  6 10 15

> cumsum(5:1)
[1]  5  9 12 14 15

> v5
[1] 3 3 3 3 1 2 3 4 7
> cumsum(v5)
[1]  3  6  9 12 13 15 18 22 29
```

The vector function `unique`, eliminates "redundant" elements in a vector and returns a vector of elements with no repeated values.

```
> unique(v5)
[1] 3 1 2 4 7

> c(rep(3,5), rep(4,10))
 [1] 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4
> length(c(rep(3,5), rep(4,10)))
[1] 15

> unique(c(rep(3,5), rep(4,10)))
[1] 3 4
> length(unique(c(rep(3,5), rep(4,10))))
[1] 2
```

The function `round` with no second argument rounds values to integers. With a second argument, it rounds to the requested number of digits.

```
> round(2.5)
[1] 2
> round(3.5)
[1] 4
> round(5/(1:3))
[1] 5 2 2
> round(5/(1:3), 3)
[1] 5.000 2.500 1.667
```

The functions `floor` (to round down) and `ceiling` (to round up) work similarly. You should explore them, using the vectors just shown to illustrate `round`.

Comparisons and logical values.

If two vectors are compared element by element, the result is a *logical* vector, which has elements `TRUE` and `FALSE`. Common comparison operators are `==` (equal), `<` (less than), `<=` (less than or equal to), `!=` (not equal) and so on. Here are some examples.

```
> 1:5 < 5:1
[1]  TRUE  TRUE FALSE FALSE FALSE
> 1:5 <= 5:1
[1]  TRUE  TRUE  TRUE FALSE FALSE
> 1:5 == 5:1
[1] FALSE FALSE  TRUE FALSE FALSE

> 1:4 > 4:1
[1] FALSE FALSE  TRUE  TRUE

> 1:5 < 4
[1]  TRUE  TRUE  TRUE FALSE FALSE

> w4
[1] 4 9 4 0
> y <- (w4 == 4)
> y
[1]  TRUE FALSE  TRUE FALSE
```

If R is forced to do *arithmetic* on logical values, then it takes `TRUE` to be 1 and `FALSE` to be 0.

```
> mean(y)
[1] 0.5

> sum(c(T, T, F, F, T, T))
[1] 4
> mean(c(T, T, F, F, T, T))
[1] 0.6666667
```

Comparisons can be used inside brackets to specify particular elements of a vector. In such instances it is convenient to read `[ ]` as "such that."

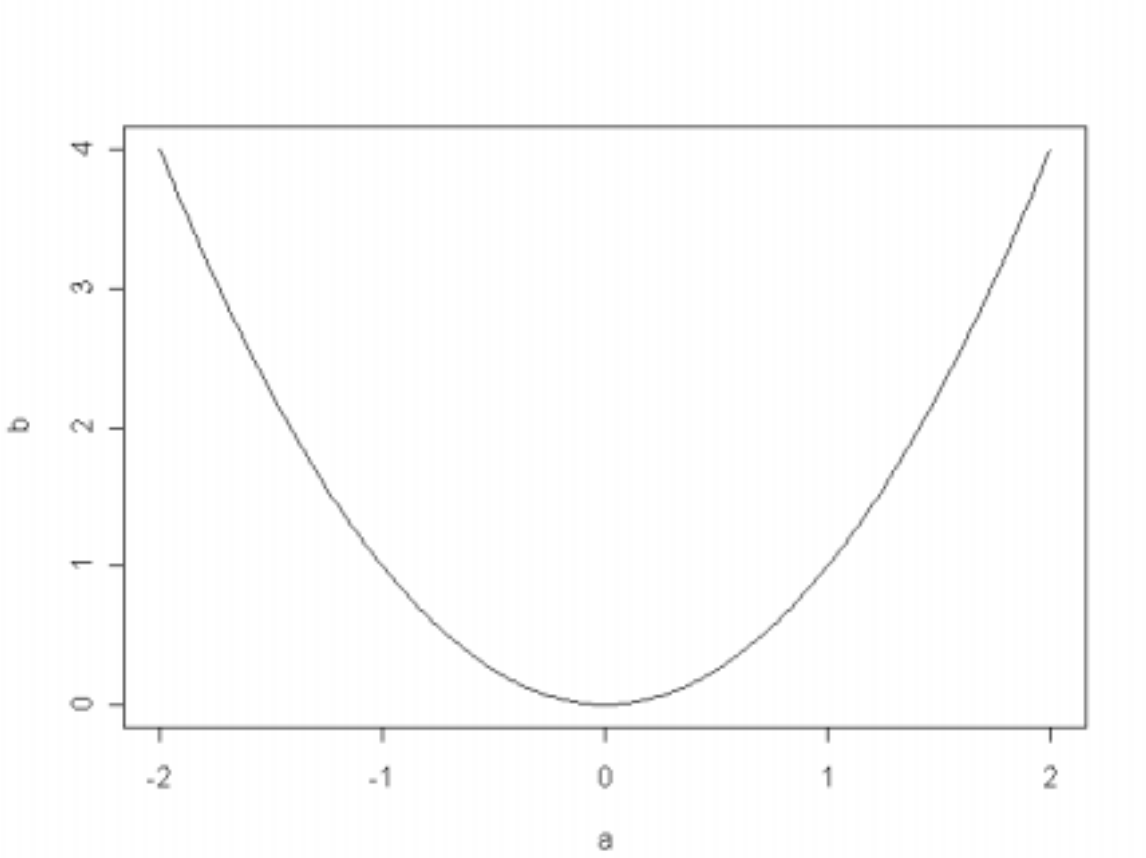
```
> v5
[1] 3 3 3 3 1 2 3 4 7
> v5[v5 < 3]
[1] 1 2
> length(v5[v5 < 3])
[1] 2
```

The last result above is logically equivalent to `sum(v5 < 3)`. Either way, we have determined that two of the elements of `v5` are smaller than 3.

Graphical functions.

R has extensive graphical capabilities. First, we illustrate the `plot` function. It is often used to make a 2-dimensional plot of one vector against another. The following code produces the graph of a parabola.

```
> a <- seq(-2, 2, length=200)
> b <- a^2
> plot(a, b, type="l")
```

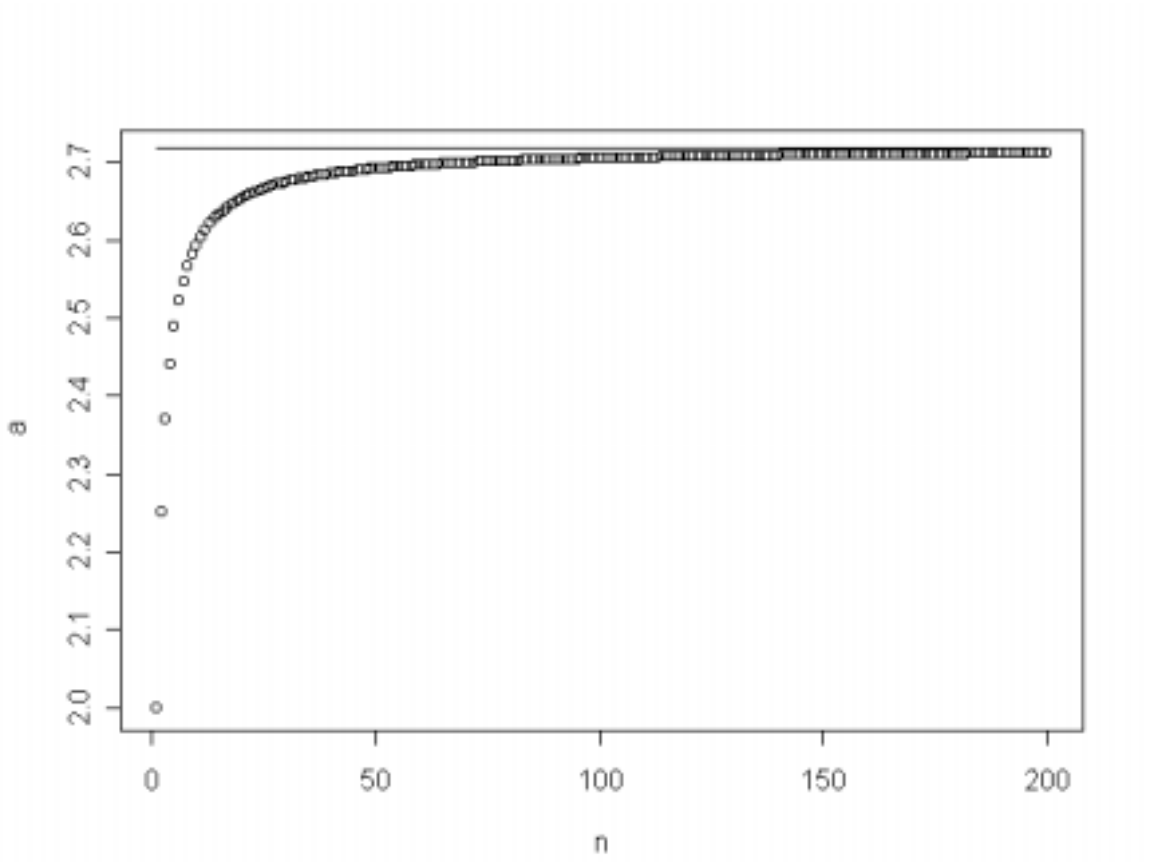


The first argument of `plot` is shown on the *x*-axis of the graph and the second is shown on the *y*-axis. The third argument (with an *ell* in quotes, not a *one*) connects the plotted points with line segments. If the plotted points are close enough together, the result looks (almost) like a smooth curve. Without the third argument each plotted point would be shown as a small open circle, and there would be no connecting lines. Try it.

If the first argument is omitted, the single vector argument is plotted on the *y*-axis against the index on the *x*-axis. With `b` defined as above, what is the result of `plot(b)`?

It is shown in calculus courses that the sequence $a_n = (1 + 1/n)^n$ converges to $e = 2.71828$ as n increases to infinity through integer values. This can be illustrated graphically as shown below. The `lines` function adds a line across the top of the graph from (1, e) to (200, e) to the plot, using the *same scale* as the plot, where exp(1) = e.

```
n <- 1:200
a <- (1 + 1/n)^n
plot(n, a)
lines(c(1,max(n)), c(exp(1),exp(1)))
```



What would be the result of substituting `plot(a, pch=".")` for the `plot` statement above? (The argument `pch="."` changes the plotting character from the "default" open circle to a small dot.) What is the value of the element `a[200]`? Make a graph similar to the one above to illustrate that the sequence $b_n = (1 + 2/n)^n$ converges to e^2.

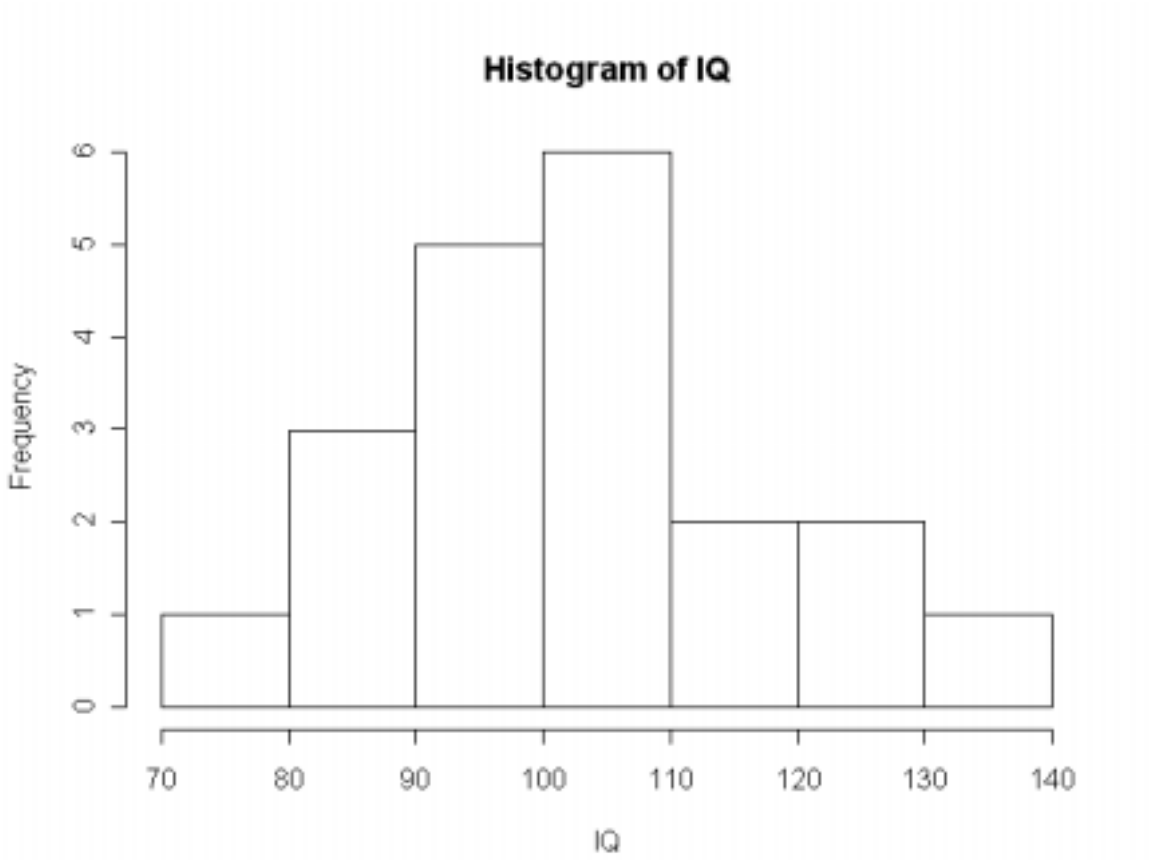
If you want to use a graph in a report, first adjust the R-Graphics window so that its size and its vertical-to-horizontal ratio is about what you want use. The graph can be saved in a variety of formats. Alternatively, it can be cut/pasted from the R-Graphics window directly into a word processor such as MS Word. (In our experience with MS Word, it is usually easiest to paste a graph into a blank paragraph that lies between two existing paragraphs.)

Another important graphical function is the histogram function `hist`. As an illustration of it, suppose the IQ scores of 20 elementary school students are listed in the vector `IQ` below. (There are cleverer ways to import data into R, but this way is simple to understand and will suffice for now.)

```
IQ <- c( 84, 108,  98, 110,  86, 123, 101, 114, 121, 131,
         90, 108, 105,  93,  95, 102, 119,  98,  94,  73)
```

Then we could make a histogram of these 20 IQ scores as follows.

```
hist(IQ)
```



You should look at the entries in the data vector and determine for yourself that there is indeed one observation in the 70s, there are three in the 80s, and so on, to verify the heights of each of the histogram bars. A histogram should appear to "balance" at the sample mean (considering the bars to have "weights" in proportion to their heights). Use `mean(IQ)` to find the sample mean of these data. (*Answer:* 102.65.)

An algorithm in R determines how many intervals will be used and what cut-points (or breaks) will separate the intervals. For most kinds of data, as above, this algorithm gives reasonable results. But the `hist` function can take additional arguments that allow you to control how the histogram is drawn. (Still more arguments can be used to shade bars, to include labels, and for other embellishments.)

Sampling from a finite population.

The `sample` function selects a random sample of a specified size from a given population. Its first argument is a vector whose elements are the *population.* Its second argument is the *sample size.* If sampling is to be done with replacement, a third argument `repl=T` is required, otherwise sampling is without replacement, (When sampling without replacement, the sample size can't exceed the population size.) We consider two specific examples.

Cards. Suppose the population is the 52 cards of a standard deck, and we want to select 5 cards at random (without replacement, of course). The following code does the sampling. But this is a random process, so your answer will (almost surely) not be the same as the one shown below.

```
h <- sample(1:52, 5)
> h
[1] 36 10 14 39 26
```

In order to view the output as actual playing cards, you would have to make a numbered list of the cards in a standard deck. If, for example, the numbers 1, 2, 3, 4 correspond to the four Aces (A♣, A♦, A♥, A♠), then there are no Aces in the 5-card hand shown above. This could be determined in R by the function `sum(h < 5)`. In the printout above, what are the elements of the logical vector `(h < 5)`? What is returned by `sum(h < 5)` in this instance? If you repeat this random experiment several times in R, you will find that you do get aces in your hand sometimes. What are the possible values that *might* be returned by `sum(h < 5)`? Using combinatorial methods, can you figure out the probability of getting no aces in a 5-card hand? What would happen if you tried to sample 60 cards? Try it in R and see,

Dice. The code below simulates rolling two dice. For each roll, the population is 1:6. Because it is possible that both dice show the same number, we are sampling with replacement.

```
> d <- sample(1:6, 2, repl=T)
> d
[1] 6 6
```

In our run of this code in R, it happens that we got two 6s. If you repeat this experiment several times, you will find that you do not usually get the same number on both dice. What is the probability of getting the same number on both dice? You could determine whether this happens on a particular simulated roll of two dice with the code `length(unique(d))`. What would this code return in the instance shown above? What are the possible values that might be returned?

R also has functions for sampling from various probability distributions. These all begin with the letter `r` followed by the abbreviated name of the distribution. For example, `rbinom(10, 5, .5)` returns a vector of length 10: the Heads counts from 10 repetitions of an experiment in which 5 fair coins are tossed.

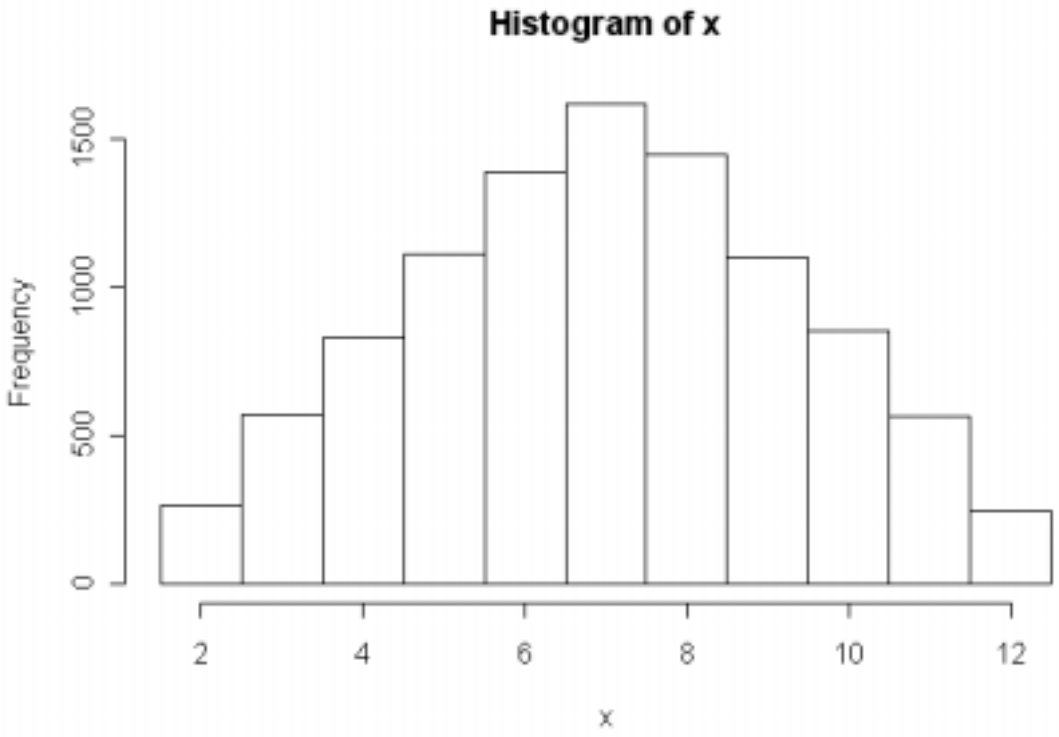
A simple loop.

Because R handles vectors as objects, many computations that would require loops in other software do not require loops in R. However, loops are convenient or necessary for some computations.

Here is a program with a loop. It simulates the distribution of the random variable X = "Sum of two dice rolled independently" by repeating the 2-dice experiment 10,000 times. (For data with only a few integer values, the algorithm in R that picks histogram break points does not give results we liked, so we used our own break points—in the vector `cut`.)

```
m <- 10000
x <- numeric(m)
for (i in 1:m){
   x[i] <- sum(sample(1:6, 2, repl=T))
   }
cut <- (1:12) + .5
hist(x, breaks=cut)
```

Notes: At the start, the vector `x` has all 0 elements. Within the loop, each element in turn is changed to an observed value of the random variable X. When the loop is complete, `x` contains 10,000 realizations of X, which are then plotted in a histogram

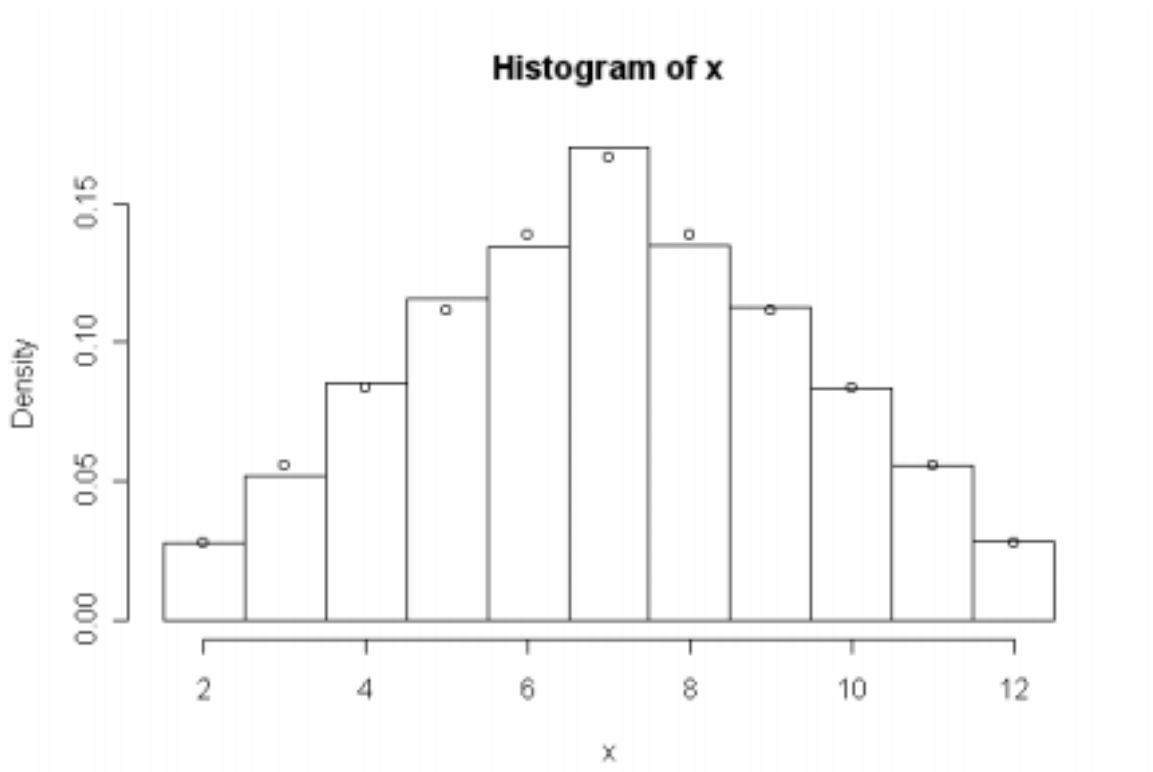


Find the distribution of the random variable X, use P(X = 4) to predict how tall bar 4 in this histogram is expected to be, and compare with the actual result (as nearly as you can read it from the graph). Then make your own histogram, which will be slightly different, and use `sum(x==4)` to find the exact count for bar 4. Do the same for bars 7 and 8. Evaluate E(X). How well is it approximated by `mean(x)`? (*Answers for 4:* 10000(3/36) = 833.33; approximated as 830 in *our* histogram; E(X) = 7, approximated as 7.0062.)

For an overall view of how close our simulation of the distribution of *X*, shown in the histogram above, comes to the actual distribution, we can plot the theoretical and simulated values on the same graph. To do this, we make two changes in the R code we used above. The last line of code above (with hist) is replaced by the three lines shown below.

- It is best to use the relative frequency or "probability" scale for the y-axis of the histogram, which is accomplished by including the argument `prob=T` in the `hist` function. (The default label on the vertical axis becomes "Density" instead of "Frequency" as before.)
- We plot the theoretical values on the histogram as open circles. This is done by using the `points` function. You should verify that the vector `p` below gives the theoretical probabilities corresponding to *x*-values `2:12`.

```
hist(x, breaks=cut, prob=T)
p <- c(1:6, 5:1)/36
points(2:12, p)
```



The agreement seems fairly good. This is a different simulation run than the one shown previously. Try several simulation runs on your own. Increasing the number of simulated rolls of the two dice to 100,000 should give a noticeably more accurate approximation, but on some computers the larger simulation may take more than a few seconds to run.

Note: It is possible to simulate this distribution by using vector addition instead of a loop, which gives a negligible running time even for 500,000 repetitions. Instead of the loop use:

```
x1 <- sample(1:6, m, repl=T)
x2 <- sample(1:6, m, repl=T)
x = x1 + x2
```

Discrete probability distributions.

R can compute probabilities from discrete and continuous probability distributions. Here we look briefly at three discrete distributions. The names of R functions that give values of a probability distribution function begin with the letter `d`. Here are some examples.

The probability of getting exactly 7 Heads in 20 tosses of a fair coin is found as follows.

```
> dbinom(7, 20, .5)
[1] 0.07392883
```

The first argument of the function is the number of Heads requested, the second is the number of coin tosses, and the third is the probability of Heads on any one toss.

Similarly, we find the probability of seeing exactly ten 6s in sixty rolls of a fair die.

```
> dbinom(10, 60, 1/6)
[1] 0.1370131
```

The distribution of the number X of heads in 10 tosses of a fair coin is a vector with 11 elements can be printed out as follows.

```
> dbinom(0:10, 10, .5)
 [1] 0.0009765625 0.0097656250 0.0439453125
 [4] 0.1171875000 0.2050781250 0.2460937500
 [7] 0.2050781250 0.1171875000 0.0439453125
[10] 0.0097656250 0.0009765625
```

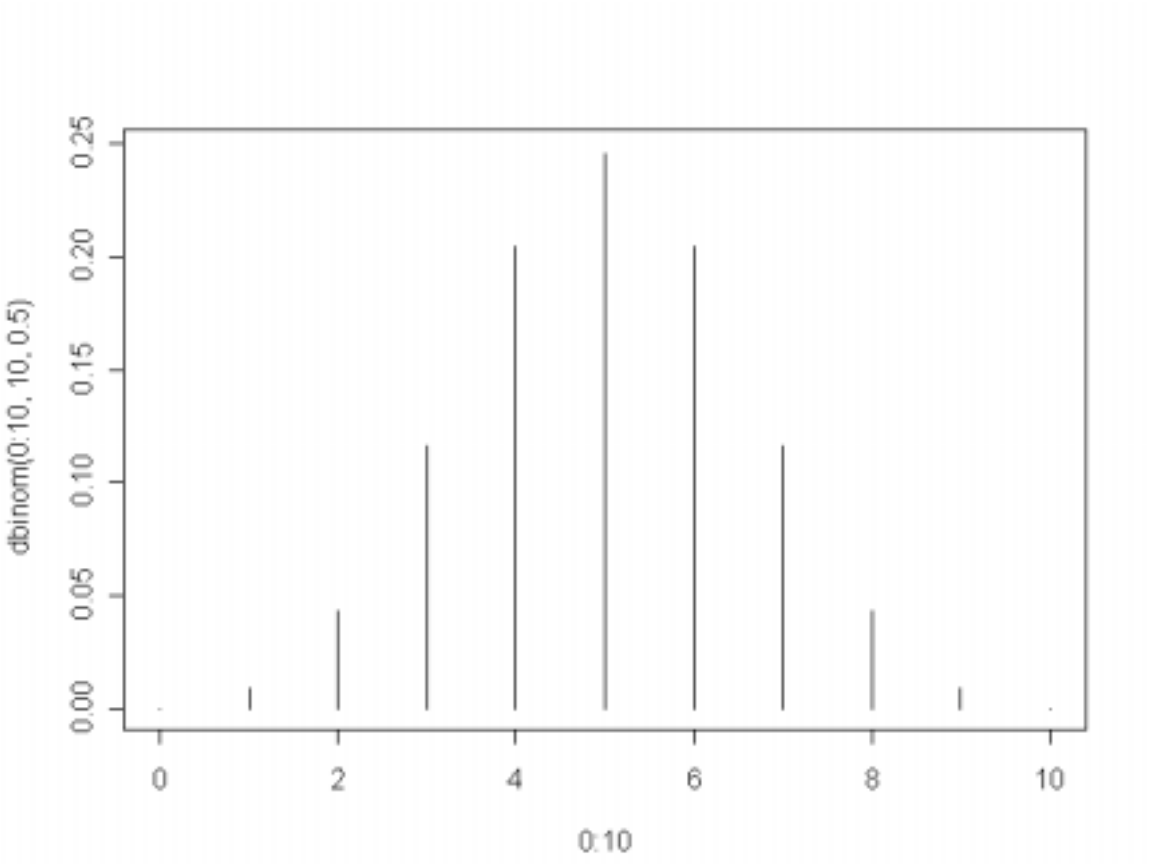
Caution: Do not be confused by the bracket notation in the printout. P(X = 0) is the element designated `[1]` in the printout above.

Here is a way to make a nicer distribution table (in which the bracketed numbers can be ignored). The function `cbind` puts column vectors together to make a matrix. We see, for example, that P(X = 3) = 0.11719, correct to five places.

```
> cbind(0:10, round(dbinom(0:10, 10, .5), 5))
      [,1]    [,2]
 [1,]    0 0.00098
 [2,]    1 0.00977
 [3,]    2 0.04395
 [4,]    3 0.11719
 [5,]    4 0.20508
 [6,]    5 0.24609
 [7,]    6 0.20508
 [8,]    7 0.11719
 [9,]    8 0.04395
[10,]    9 0.00977
[11,]   10 0.00098
```

We can make a bar chart of this binomial distribution as shown below. The argument `type="h"` produces the vertical bars. (Without this argument, the default would be to put open circles where the tops of the bars are in the graph below; try it.)

```
plot(0:10, dbinom(0:10, 10, .5), type="h")
```



In a similar way `dpois(0:10, 2)` gives the probabilities of seeing 0, 1, ..., 10 Poisson events in a model where the mean number of events is $\lambda = 2$. Also, `dhyper(0:4, 4, 48, 5)` gives the probabilities of seeing 0, 1, 2, 3, or 4 Aces in a poker hand. (There are 4 Aces and 48 non-Aces in the deck of which 5 are drawn at random without replacement for form the hand.) Use R to make distribution tables for these two distributions.

To get *cumulative probabilities* $P(X \le x)$, use the functions `cbinom`, `cpois`, and `chyper`. Use these functions to make cumulative distribution tables for the specific binomial, Poisson, and hypergeometric distributions mentioned above.