1 Part I

1.1

R knows six different vector types, namely: logical, integer, real, complex, character (string) and raw. To give some examples for every type:

```
> # define logical object
> log <- TRUE
> is.logical(log)
```

```
[1] TRUE
```

```
> # define integer object
> int <- 1:5
> is.integer(int)
```

```
[1] TRUE
```

```
> # define real (numeric double) object
> real <- 2.5
> is.double(real)
```

```
[1] TRUE
```

```
> # define complex object
> comp <- 1+2i
> is.complex(comp)
```

```
[1] TRUE
```

```
> # define character (string) object
> char <- "a"
> is.character(char)
```

```
[1] TRUE
```

```
> # define raw object
> rawd <- as.raw(22) # corresponds to 16
> is.raw(rawd)
```

```
[1] TRUE
```

1.2

Difference between generic and numeric vector:

- An *atomic* vector contains only one single "atomic" data type in all entries. An example would be a vector which contains only integers.
- A *generic* vector (like a `list`) can contain different types of data. An example would be a vector which contains characters and numbers.

1.3

To explain: *A data frame is a list, but not evey list is a data frame.*

- A `list` is an object containing collections of objects. The types of the entries inside of the list can be different. It is for example allowed that a `list` contains a vector of real values (doubles) and a vector of characters. The length of the containing vectors can be **different**.
- A `data frame` is an object containing colletions of objects. The types of the entries inside of the list can be different. The length of the vectors have to be **the same**. The `data frame` has a matrix-like structure.

`list` and `data frame` are very similar, but the `data frame` has one more restriction (same length of all vectors). That's why a `data frame` is always a list, but a `list` is not always a `data frame`.

2 Part II

For random number generation R uses pseudo-random numbers. Starting from an initial state, called *seed state*, it will produce a deterministic sequence, which is used as random numbers. If we choose the same seed in every turn, we get the same results. To make the results of random numbers comparable, we first set the seed in a sepecific state, using `set.seed`.

```
> # set seed state to specific state
> set.seed(1)
```

After setting the seed, we define a vector with (pseudo-) random values. In this case we create $1 \cdot 10^8$ random values following normal distribution. Using the function `rnorm`, we create a distribution with mean 5 and standard deviation of 10 and saving them in a vector called `largeVector`.

```
> # define vector with normal random values
> largeVector <- rnorm(1e6, mean=5, sd=10)
```

The function `cumsum` calculates the cumulative sum of the values of the vector. It takes all elements one by one and calculates for this element the sum of all elements before, including the current element. These values will be the new elements of the new vector. Consider following example:

$$\begin{pmatrix} 1 \\ 4 \\ 3 \end{pmatrix} \xrightarrow{\texttt{cumsum}} \begin{pmatrix} 1 \\ 5 \\ 8 \end{pmatrix}$$

In the the first line of the follwing snippet, it first calculates the `cumsum` of the whole vector `largeVector`. Afterwards it just takes the first 100 elements and saves them in vector `a`. In the second line, it first takes the 100 first elements

of `largeVector` and calculates the `cumsum` afterwards, which is saved in vector `b`. The second line should be much faster (see below), even if the results is the same (see also below).

```
> # get cumulative sum of the first 100 elements of largeVector
> a <- cumsum(largeVector)[1:100]
> b <- cumsum(largeVector[1:100])
```

As mentioned before, the results of vectors `a` and `b` should be the same. To check if all elements of the two vectors are exactly equal, we can use the function `identical`, where the result is `TRUE`.

```
> # check if both methods lead to exactly same result
> identical(a,b)

[1] TRUE
```

In the next step, we can compare the speed of the two ways to calculate the vectors `a` and `b`. To check the elapsed time while calculating we can use the function `system.time`, which gives us the CPU caluclation time.

```
> # get CPU calulation time of first method
> system.time(cumsum(largeVector)[1:100])

   user  system elapsed
  0.008   0.000   0.008

> # get CPU calulation time of second method
> system.time(cumsum(largeVector[1:100]))

   user  system elapsed
      0       0       0
```

The *user* CPU time and the *system* CPU time is a technical distinction in time running the R code and time used in operating system kernel on behalf of the R code. The interesing time is the *elapsed* time, which is the sum of the *user* time and the *system* time. We can see that the first operation of taking the `cumsum` of the whole `largeVector` with 100 million elements (and reducing the vector to 100 elements afterwards) takes a lot more CPU calucaltion time than taking the `cumsum` of the first 100 elements directly. The second method is much more efficient than the first method, because in the end we are only interested in the `cumsum` of the first 100 elements of the vector.

3 Part III