

Федеральное государственное образовательное бюджетное учреждение  
высшего профессионального образования  
«ФИНАНСОВЫЙ УНИВЕРСИТЕТ  
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»  
(Финансовый университет)

---

Департамент анализа данных, принятия решений  
и финансовых технологий

Дисциплина «Программирование в среде R»

П.Б. Лукьянов

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ  
ЛАБОРАТОРНОЙ РАБОТЫ № 4

**Векторы и операции с ними**

Для студентов, обучающихся по направлению подготовки  
«Прикладная математика и информатика»  
(программа подготовки бакалавра)

Москва 2020

Для быстрой и эффективной обработки больших массивов информации простейших типов данных (целые и действительные числа, строки, логические значения TRUE / FALSE) оказалось недостаточно. В R определены более сложные структуры данных, для работы с которыми используются специальные функции.

Цель лабораторной работы заключается в изучении самого распространенного контейнера для хранения данных – вектора.

## 1. Понятие вектора

Вспомним математику. В школьной математике используются две большие категории данных: скалярные величины и векторы. Скаляром называют величину, которая имеет только одну характеристику – численное значение. Все операции с числами – это скалярные операции.

Вектор – математический объект, имеющий кроме величины еще и направление. Если в пространстве задана система координат, то вектор однозначно задаётся набором своих координат. Поэтому в математике, информатике и программировании упорядоченный набор чисел часто тоже называют вектором.

Дадим следующее определение вектору. **Вектор – это объект, объединяющий элементы одного типа.** Вектор предназначен для хранения данных, состоящих из нескольких однотипных значений. Сочетания разных типов данных в векторе не допустимы. Любой вектор характеризуется типом хранимых данных и своей длиной. Тип вектора `x` проверяется функциями `typeof(x)`, `mode(x)`; длину вектора можно узнать, вызвав функцию `length(x)` (см. рис. 1).

Например, в одном векторе может храниться возраст всех студентов группы, в другом векторе – их имена и фамилии, в третьем векторе – средний бал успеваемости по каждому студенту, в четвертом векторе – номера их телефонов.

Вектор



Вектор – именованный одномерный объект, содержащий набор однотипных элементов (или числовые, или логические, или текстовые значения).

Тип вектора **x** проверяется функциями **typeof(x)**, **mode(x)**

Длина вектора проверяется функцией **length(x)**

Элементы вектора проиндексированы, к каждому элементу вектора можно обратиться по его индексу: **x[i]**

Рис. 1. Вектор – структура для хранения данных одного типа

В R вообще нет скалярных величин. Следовательно, изначально в R все создаваемые объекты – уже векторы. Таким образом, при изложении предыдущего материала мы рассматривали и создавали векторы, состоящие из одного значения. Принципиальное решение создателей языка о том, что R будет работать только с векторами, связано с тем, что задачи обработки информации решаются проще, если все данные хранить и представлять векторами.

В соответствии с этим подходом в R были разработаны функции, реализующие быстрые векторные вычисления для обработки данных и представления результатов. Как следствие, традиционные решения программистов с использованием циклов для перебора значений в R не

используют, так как циклы по сравнению с векторными операциями работают гораздо медленнее.

## 2. Создание векторов

Создадим вектор `w`, в котором будут храниться четные числа от 1 до 10. Для создания такого вектора вызовем функцию `c()`:

```
w <- c(2, 4, 6, 8, 10)
```

Функция `c()` (от слова *concatenate* – связывать) создает объект «вектор». Таким образом, нами был создан вектор `w` из 5 элементов. Как узнать, какому типу принадлежат элементы `w`?

Создать вектор можно, используя разные типы данных, но все данные внутри одного вектора будут приведены к одному типу:

```
w1 <- c(-12L, 4, 'Level 2', 8.2, TRUE)
```

```
w2 <- c(0, 56/6, FALSE, FALSE, 18)
```

```
w3 <- c(Inf, NULL, NA, FALSE, 18, NaN)
```

```
w4 <- c(Inf, NULL, NA, FALSE, 18L, NaN)
```

```
w5 <- c(Inf, NULL, NA, FALSE, '18', NaN)
```

```
w6 <- c(NULL)
```

Какого типа вектора `w1`, `w2`, `w3`, `w4`, `w5`, `w6`? Чему равна их длина? Почему векторы `w3` и `w4` одного типа?

Создать вектор `z` с пустыми (нулевыми) значениями можно с помощью функции `vector()`:

```
z <- vector()
```

Выясните, какого типа и какой длины был создан вектор `z`.

Если требуется вектор-заготовка для хранения данных определенного типа и определенного размера, вызывают функцию `vector()` с заданием нужных параметров:

```
z2 <- vector(mode="numeric", length=129)
```

```
z3 <- vector(mode="logical", length=22)
```

```
z4 <- vector(mode="integer", length=34)
```

```
z5 <- vector(mode="character", length=0)
```

```
z6 <- vector(mode="double", length=19)
```

Обратите внимание, при создании векторов мы используем инициализацию именованных параметров, и форма записи может быть другой:

```
z6 <- vector(length=19, mode="double")
```

```
z7 <- vector(length=19)      # тип данных задается по умолчанию
```

```
z8 <- vector(mode="double") # длина вектора задается по умолчанию
```

Например, вектор `z4` предназначен для хранения целых значений.

Выясните, к каким типам принадлежат вектора `z2`, `z7`. Какова длина `z8`?

Создание векторов определенного типа можно упростить:

```
z2 <- numeric(length=129)
```

```
z3 <- logical(length=22)
```

```
z4 <- integer(length=34)
```

```
z5 <- character(length=0)
```

Чтобы узнать, что содержится в объекте `w`, достаточно ввести `w` и нажать Enter. Проверьте, какие вектора будут созданы при выполнении команд

```
w <- c(10:3)
```

```
w <- c(1:20, 3)
```

```
w <- c(10:5, 5:10, 15:18)
```

Вектор из последовательности чисел с шагом 1 можно создать еще проще:

```
w <- -4:10
```

Обратите внимание, что границы диапазона значений не обязательно должны быть целочисленными. Какой вектор получится, если задать границы действительными числами?

Важно! Оператор `:` имеет приоритет над арифметическими операторами и выполняется первым:

```
> 1:10-1          # набрали
```

```
[1] 0 1 2 3 4 5 6 7 8 9 # получили результат
```

```
> 1: (10-1)        # набрали, использовали скобки, изменили приоритет
```

```
[1] 1 2 3 4 5 6 7 8 9 # результат другой
```

Если нужен вектор, состоящий из повторяющихся значений, используется функция `rep()` (от *repeat* – повторять):

```
w <- rep(TRUE, 8)
```

Можно дублировать последовательности значений; посмотрите, что будет содержать итоговый вектор при различных параметрах:

```
w <- rep(c(0, -1, 1:3), times = 3)
```

```
w <- rep(c(0, -1, 1:3), each = 3)
```

```
w <- rep(c(0, -1, 1:3), each = 3, times = 2)
```

В первом случае 3 раза дублируется весь вектор `(c(0, -1, 1:3))`, во втором случае первый элемент исходного вектора дублируется три раза, затем три раза дублируется второй и т.д. В третьем случае сначала выполняется команда

```
w <- rep(c(0, -1, 1:3), each = 3)
```

а затем получившийся вектор дублируется `times` раз:

```
w2 <- rep(w, times = 2)
```

Часто возникает задача получения вектора, содержащего некоторую последовательность значений. Функция `seq()` (от *sequence* – последовательность) решает эту задачу (см. рис. 2).

В функции `seq()` у каждого параметра есть значение по умолчанию. Определите эти значения для параметров `to`, `from`, `by`. Еще один параметр функции `seq()` задает общее количество элементов последовательности, это параметр `length.out`:

```
w <- seq(4, from=12, length.out = 10)
```

```

>
> w <- seq(from = -11, to = 5, by = 2.5)
> w
[1] -11.0 -8.5 -6.0 -3.5 -1.0  1.5  4.0
> w <- seq(to = -11, from = 5, by = -2.5)
> w
[1]  5.0  2.5  0.0 -2.5 -5.0 -7.5 -10.0
>
> |

```

Рис. 2. Разбиение отрезка начинается от значения «from»

Использование при вызове функции всех четырех параметров приведет к ошибке:

```
w <- seq(from = -11, to = 5, by = 2.5, length.out = 22)
```

Задание. Приведите пример создания вектора через вызов функции `seq()` в случае одновременного использования параметров `by` и `length.out`. Объясните правило формирования вектора в этом случае.

Следует отметить, что при вызове функции `seq()` используется другая функция, `seq_len()`. Самостоятельно исследуйте параметры и поведение `seq_len()` и объясните, в каком случае вызов `seq()` приведет к вызову и исполнению `seq_len()`.

### 3. Считывание вектора с клавиатуры или из файла

Порой проще создать нужный вектор, считывая вводимые значения с клавиатуры или из файла данных напрямую. Этот подход реализован в функции `scan()`. У этой функции более двадцати (!) параметров; если запустить `scan()`, не задав ни одного параметра, будет реализован следующий сценарий:

- активируется режим считывания действительных чисел с клавиатуры

- Пользователь вводит число, нажимает Enter, число заносится в элемент вектора
- Пользователь может набрать несколько действительных чисел в строке, разделяя их произвольным количеством пробелов, все числа последовательно займут соответствующие элементы вектора
- Считывание чисел выполняется до тех пор, пока Пользователь не нажмет Enter два раза (рис. 3).

Обратите внимание, что вместо одного из отсутствующих значений было введено NA; был использован маркер Inf, говорящий об уходе значения за пределы шкалы измерения.

Каков смысл параметров функции `scan()`? Вся необходимая информация содержится в подсказках. Разберите назначение параметров функции, для чего изучите подсказку, набрав `?scan`.

С какими параметрами нужно запустить `scan()`, чтобы был реализован ввод с клавиатуры элементов логического вектора?

Как реализовать ввод вектора из строк?

```
>
> scan()
1: -9.9
2: 0
3: 223.009
4: 12 7 7          Inf 0  5 5      NA -72.007 0 0
15: 19
16: 220000 -4.4
18:
Read 17 items
[1] -9.900  0.000 223.009 12.000
[5]  7.000  7.000      Inf  0.000
[9]  5.000  5.000      NA  -72.007
[13]  0.000  0.000 19.000 220000.000
[17] -4.400
>
```

Рис. 3. Ввод данных с использованием функции `scan()`



Один из недостатков использования `scan()` заключается в том, что при вводе данных с помощью этой функции легко сделать ошибку без возможности ее исправления. Для редактирования содержимого вектора предназначена функция `fix()`, где единственный обязательный параметр – это имя вектора. `fix()` запускает простейший редактор, в котором можно делать любые исправления, а результат сохранить (рис. 4).

Фактически при своем вызове функция `fix()` вызывает функцию `edit()` для редактирования переданного в качестве параметра вектора, и можно было бы заменить вызов `fix(имя_вектора)` вызовом `edit(имя_вектора)`, так как редактор используется один и тот же (рис. 4).

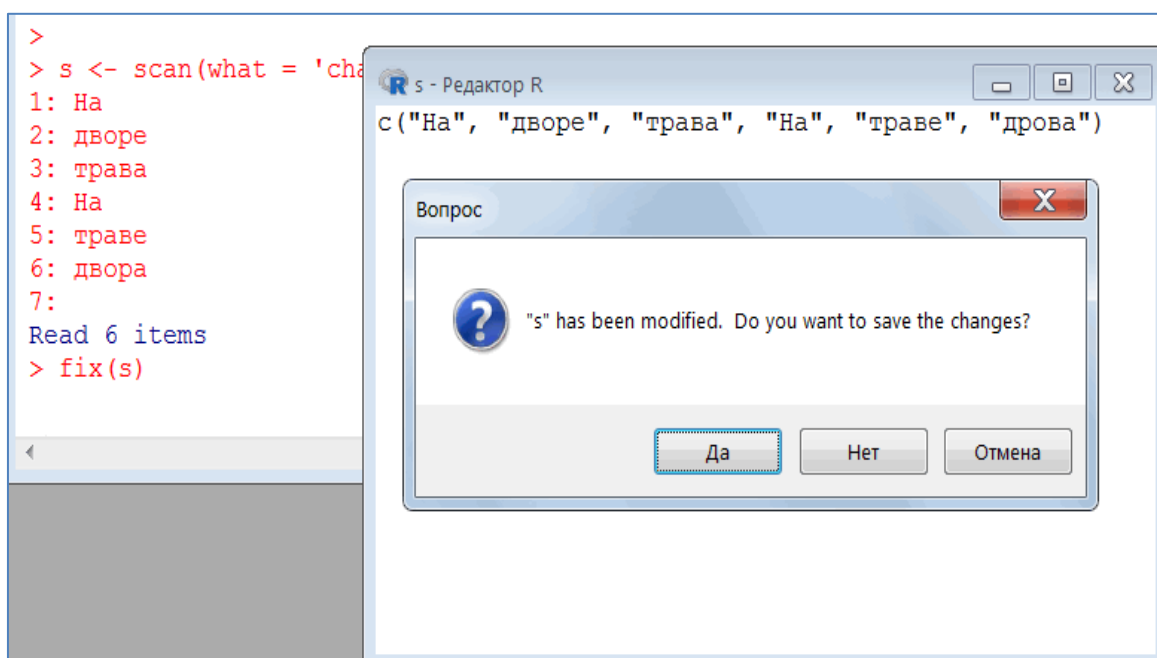


Рис. 4. Коррекция содержимого вектора функцией `fix()`

Но в работе этих функций есть некоторые отличия. Сравните редактирование с помощью `fix()` и с помощью `edit()` и выясните, в чем заключаются различия.

#### 4. Генерация случайных значений и сортировка

Частая задача при обработке данных – выполнить сортировку вектора в ту или иную сторону. Рассмотрим, как получить вектор из случайных значений в некотором диапазоне, чтобы затем эти значения отсортировать. Генерация случайных значений действительных чисел выполняется вызовом функции `runif()` (см. рис. 5). Обязательный параметр – количество случайных чисел `n`. Если задать `n=100`, получим вектор из 100 случайных значений.

Если других параметров у `runif()` нет, случайные числа создаются в диапазоне от 0 до 1. Для задания диапазона, отличного от (0, 1), используется задание фактических значений у параметров `min` и `max`.

```
>
> x <- runif(n=1)
> x
[1] 0.358101
>
> x <- runif(4)
> x
[1] 0.65999757 0.09373953 0.85173417 0.69833657
>
> x <- runif(-12.3, n= 5, max = -5.05)
> x
[1] -8.536526 -9.358172 -6.707490 -8.395607 -8.131678
>
> |
```

Рис. 5. Генерация случайных значений функцией `runif()`

Для получения случайного **целого числа** используется функция `sample()`, параметры которой задают диапазон (например, `x=1:100`), количество случайных значений (например, `size = 22`), запрет или разрешение появления повторных случайных значений (`replace=FALSE / TRUE`) (см. рис. 6).

Сортировка вектора выполняется функцией `sort()`. Функция `sort(имя_вектора)` сортирует элементы вектора по возрастанию, вызов функции `sort()` со значением параметра `decreasing=TRUE` сортирует вектор в обратную сторону (рис. 7).

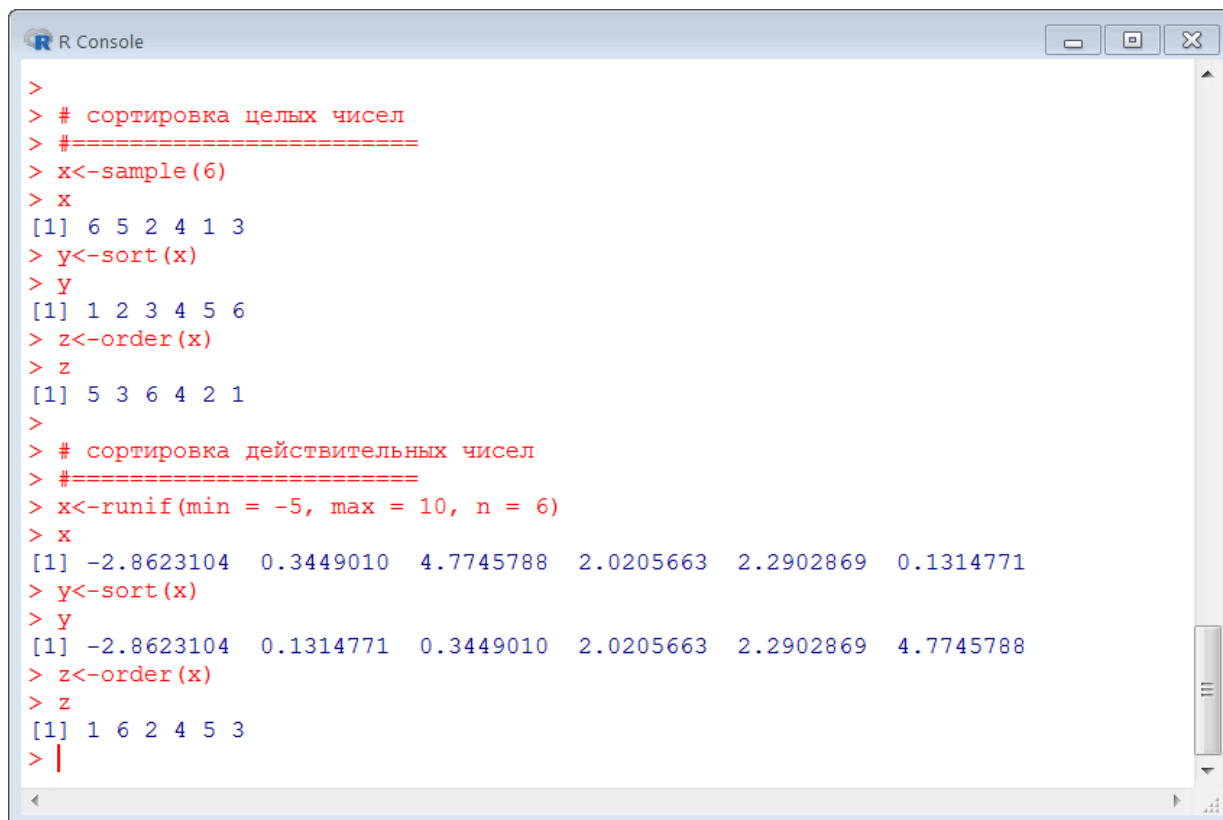
```
> x<-sample(8)
> x
[1] 8 5 1 7 6 4 3 2
>
> x<-sample(4:8)
> x
[1] 7 4 8 5 6
>
> x<-sample(-3:5, 12)
Ошибка в sample.int(length(x), size, replace, prob) :
  не могу сделать выборку большую чем популяция если 'replace = FALSE'
> x
[1] 7 4 8 5 6
>
> x<-sample(-3:5, 12, replace=TRUE)
> x
[1] -2  5  5 -2 -2  0  4 -2  4  1  3 -3
>
>
> |
```

Рис. 6. Генерация случайных значений функцией `sample()`

```
> x <- runif(-100, n= 6, max = 0)
> x
[1] -61.44684 -45.08358 -13.74772 -43.00052 -58.59973 -75.46657
>
> y<- sort(x)
> y
[1] -75.46657 -61.44684 -58.59973 -45.08358 -43.00052 -13.74772
>
> y<- sort(-x)
> y
[1] 13.74772 43.00052 45.08358 58.59973 61.44684 75.46657
>
> y<- sort(-x, decreasing = TRUE)
> y
[1] 75.46657 61.44684 58.59973 45.08358 43.00052 13.74772
>
> |
```

Рис. 7. Работа функции `sort()`

Кроме `sort()` задачи сортировки решает функция `order()`, но в отличие от `sort()` результатом является вектор индексов, значения которых дадут отсортированную последовательность (см. рис 8).



```
>
> # сортировка целых чисел
> #=====
> x<-sample(6)
> x
[1] 6 5 2 4 1 3
> y<-sort(x)
> y
[1] 1 2 3 4 5 6
> z<-order(x)
> z
[1] 5 3 6 4 2 1
>
> # сортировка действительных чисел
> #=====
> x<-runif(min = -5, max = 10, n = 6)
> x
[1] -2.8623104  0.3449010  4.7745788  2.0205663  2.2902869  0.1314771
> y<-sort(x)
> y
[1] -2.8623104  0.1314771  0.3449010  2.0205663  2.2902869  4.7745788
> z<-order(x)
> z
[1] 1 6 2 4 5 3
> |
```

Рис. 8. Отличия в работе функций `sort()` и `order()`

## 5. Манипуляции с векторами

С векторами можно делать множество преобразований, рассмотрим некоторые из них. Как добавить элементы в вектор? Используется функция `append()`:

```
w<- append(3:12, -2:0) # вектор 3:12 будет увеличен на 3 элемента
```

Формальные параметры у `append()` следующие:

`x` – исходный вектор

`values` – вектор, который должен быть добавлен

`after` – номер индекса, после которого будет добавлен `values`

Если параметр `after` не указан, добавление выполняется после последнего индекса, `x` и `values` – обязательные параметры. Если мы работаем с вектором `x` и хотим добавить в него несколько элементов, вызов `append()` может быть таким:

```
x <- append(x=x, values=x[4:6], after=length(x)-2)
```

В приведенном примере важно понимать, что означает каждое использование символа `x`:

- `x` в крайней левой позиции задает имя итогового вектора, который получится в результате добавления элементов в исходный вектор `x`, т.е. новому вектору дается имя старого вектора
- `x` в левой части равенства `x=x` – это имя формального параметра функции `append()`, задающего исходный вектор, к которому выполняется добавление
- `x` в правой части равенства `x=x` – это имя исходного вектора, к которому добавляются новые элементы
- `x[4:6]` в выражении `values=x[4:6]` означает, что добавляться будут три элемента исходного вектора `x`, расположенные в ячейках с 4-й по 6-ю
- `length(x)` в выражении `after=length(x)-2` означает длину исходного вектора `x`. Добавляться три новых элемента будут после элемента исходного вектора `x` с номером `length(x)-2`

Если к вектору прибавить число, то оно прибавится ко всем элементам вектора:

```
x <- 6 + c(1, 5:7) # получим (7, 11, 12, 13)
```

Аналогично, умножение, деление и все математические формулы с векторами применяются ко всем элементам вектора (см. рис. 9).

```

>
> x<-sample(4:8)
> x
[1] 6 5 8 4 7
> y<-log10(abs(28 * sin(x) / cos(x)) / sqrt(x))
> y
[1] 0.5219846 1.6266559 1.8281035 1.2097696 0.9648505
> z<- x^(sqrt(x))
> z
[1] 80.55148 36.55480 358.36387 16.00000 172.15484
>
> |

```

Рис. 9. Пример использования вектора в формулах

Именно в таком подходе заключается основное преимущество R – если одни и те же действия нужно выполнить над тысячами значений, эти значения помещаются в вектора, а затем пишется одна формула его обработки.

Два вектора можно объединить в третий вектор (см. рис. 10). Обратите внимание, что при объединении векторов разного типа преобразование типов выполняется автоматически. Какими функциями проверяется тип вектора?

```

>
> x1<- 1:5
> x2<--4:-6
> x3<-c(x2, x1)
> x3
[1] -4 -5 -6 1 2 3 4 5
>
> y1<-c('Когда', 'прилетит', 'вертолет?')
> y2<-12.40
> y3<-c(y1, y2)
> y3
[1] "Когда" "прилетит" "вертолет?" "12.4"
>
>

```

Рис. 10. Примеры объединения векторов

Как убрать лишние элементы вектора? Как получить подмножество элементов вектора? Для этого используются квадратные скобки с указанием нужного диапазона:

```
x <- x[3:5]
```

```
z <- c(x[3:5], x[7], x[1:4]) # некоторые элементы войдут в z два раза
```

Функцию конкатенации можно помещать внутрь индексации, выделяя отдельные элементы векторов:

```
z <- x[c(2,4,6)]
```

```
z <- c(x[c(2,4,6)], y[c(length(y)-3:length(y))])
```

Для удаления элемента вектора перед его индексом ставят минус:

```
z <- x[c(-3, -2, -5)] # будут удалены 2-й, 3-й и 5-й элементы вектора x
```

Для фильтрации значений элементов вектора можно использовать логическое условие:

```
z <- x[c(x<=1, x>400)] # будут выбраны элементы, значения которых  
# меньше или равны 1 и больше 400
```

Значения векторов можно менять поэлементно:

```
x[3]<- 44.6
```

```
x[2:6]<-c(17:19, sample(1:10,2))
```

В последнем примере меняются значения вектора *x*, расположенные в индексах с 2 по 6; для второго, третьего и четвертого индексов значения будут соответственно 17, 18, 19. Оставшиеся два значения с индексами 5 и 6 будут заполнены случайными величинами из диапазона от 1 до 10.

Если в манипуляциях с индексами выйти за длину вектора, ошибки не произойдет, длина вектора увеличится, а все неопределенные элементы получат значение NA (см. рис. 11).

## 6. Сравнение векторов

Как узнать, какие элементы вектора больше или меньше какого-либо значения? Пусть имеется вектор с массами сотрудников:

```
mas <- c(55, 71, 84, 90, 77, 60, 58, 94, 49, 53, 81)
```

Есть ли сотрудники с массой более 90 кг? 100 кг? Напишем условия:

```
m100 <- mas >= 100
```

```
m90 <- mas >= 90
```

Выполните код, проверьте результаты. Если хотим узнать, какие значения масс превышают или равны 90 кг, напишем другую формулу:

```
m90 <- mas[mas>=90]
```

```
>
> x[22:26]<-c(17:19, sample(1:10,2))
> x
[1] -61.44684 17.00000 18.00000 19.00000 10.00000 6.00000 NA
[8] NA NA NA NA NA NA NA
[15] NA NA NA NA NA NA 17.00000
[22] 17.00000 18.00000 19.00000 10.00000 2.00000
>
> |
```

Рис. 11. Автоматическое увеличение длины вектора с инициализацией неопределенных элементов значениями NA

Можно сравнивать вектора между собой (см. рис. 12). В примере на рис. 5-17 вектора не только имеют разный тип, но и разный размер. При сравнении векторов было выдано предупреждение о том, что длины векторов не совпадают, но сравнение все равно было выполнено. Разберем по шагам, какие действия были сделаны средой R перед сравнением:

- Было выдано предупреждение о несовпадении размеров векторов
- Длина вектора b была увеличена на 5 элементов для того, чтобы векторы a и b были одного размера
- В эти пять новых элементов вектора b были скопированы 5 первых элементов этого же вектора



- Было произведено поэлементное сравнение векторов a и b, результат записан в вектор z

```
>
> a<- sample(1:100, size = 11)
> a
[1] 15 49 59 21 98 95 85 57 34 76 41
> b<- runif(n=6, min=1, max=100)
> b
[1] 59.192064 10.632442 88.130070 7.324676 83.313045 61.953748
> z<- a<b
Предупреждение:
В a < b :
  длина большего объекта не является произведением длины меньшего объекта
> z
[1] TRUE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE TRUE
>
> |
```

Рис. 12. Сравнение двух векторов разного типа и разной длины

На этом примере показан общий подход, реализуемый в R при операциях с векторами, имеющими разную длину: вектор, имеющий меньшую длину, дополняется до вектора с большей длиной. Новые элементы вектора заполняются значениями этого же вектора, начиная с его значения по первому индексу и далее по порядку, в цикле, пока все поля не заполнятся. Только после этого начинают выполняться действия с векторами.

Если под этим углом посмотреть на выражения, где выполняются действия между скаляром (единичным вектором) и другим вектором, то становится понятно, что на самом деле действия совершаются между векторами одинаковой длины:

$10 + c(1:5)$              $\# (10 + 1, 10 + 2, 10 + 3, 10 + 4, 10 + 5)$

$18.8 / c(3,5,7,9)$      $\# (18.8 / 3, 18.8 / 5, 18.8 / 7, 18.8 / 9)$

Сформулируем два общих вопроса, связанные с анализом результата проверки вектора на некоторое условие:

1. Удовлетворяет ли ХОТЯ БЫ ОДИН элемент вектора заданному условию? (логическое ИЛИ)
2. Удовлетворяют ли ВСЕ элементы вектора заданному условию? (логическое И)

Ответ на первый вопрос дает функция `any()`, где параметрами функции являются логические вектора. Проверим, есть ли среди сотрудников те, у кого масс меньше 50 и больше 100 кг?

```
res<- any(mas<50, mas>100)
```

`res` вернет `TRUE`, так как в векторе `mas` есть элементы, меньшие 50, первое условие истинно.

Не обязательно в логических параметрах должны быть условия, связанные с одним вектором, рассмотрим пример:

```
mas <- c(55, 71, 84, 90, 77, 60, 58, 94, 49, 53, 81)
```

```
height <- c(162,183, 174, 181, 169, 166,173,191,159, 170, 180)
```

```
res<- any(mas<50, mas>100, height >=200)
```

Для проверки истинности всех логических условий предназначена функция `all()`, где параметры функции такие же, как и у `any()`. Проверим ее работу:

```
res<- all(mas ==55, height > 190)
```

`res = FALSE`, так как не все элементы `mas` равны 55, не все значения из `height` больше 190. Даже по отдельности эти условия дадут `FALSE`:

```
res <- all(mas ==55)
```

```
res <- all(height > 190)
```

`all()` вернет `TRUE`, если все элементы `mas` и все элементы `height` будут удовлетворять условию, например:

```
res <- all(mas >=49, height < 200)
```

## Контрольные вопросы и задания

1. Исследовать и сформулировать отличия в работе функций, определяющих тип вектора: `typeof()`, `mode()`, `str()`

2. Даны два числовых вектора разной длины:

```
a<- c(7:4, 0)
```

```
b<-c(8, 10.5, 0, -2, 9)
```

Написать программу, выполняющую следующие действия над векторами:

- 1) Сложение векторов  $a + b$
- 2) Умножение векторов  $a * b$
- 3) Деление векторов  $a / b$
- 4) Нахождение среднего арифметического для каждого вектора
- 5) Нахождение суммы элементов каждого вектора
- 6) Представление результатов расчетов на экране

Указание. В программах использовать функции `print()`, `paste()`, `paste0()`, `sum()`, `mean()`.

3. Пусть вектор `w` создан из значений разного типа. Выяснить правило, по которому выполняется преобразование данных разных типов, хранящихся в векторе. Для этого последовательно создать вектора с данными одного типа, двух типов, трех типов и т.д. Перебрать все возможные сочетания типов.

4. Аналогично заданию 2, проверить работу арифметических операторов для векторов. Последовательно использовать векторы:

- 1) одного типа, одного размера
- 2) одного типа, разных размеров
- 3) разных типов, одного размера

4) разных типов, разного размера

Обобщить результаты расчетов в виде правил вычисления результирующего вектора.

5. Создать несколько векторов разных типов и длины и написать с ними три формулы с использованием логических операций. Объяснить полученный результат, добавив в скрипт комментарии.
6. Написать формулу получения 10 случайных целых чисел из диапазона ( $\min = -7$ ,  $\max = 28$ ) с помощью функции `runif()`, не используя при ее вызове задания минимального и максимального значений.
7. Написать формулу получения 20 случайных действительных чисел из диапазона ( $\min = -7$ ,  $\max = -2$ ) с помощью функции `sample()`, не используя при ее вызове задания минимального и максимального значений.
8. Написать программу получения вектора из N случайных символов русского алфавита. Использовать функцию `sample()`.