

Combinatorics

by Andri Signorell

Helsana Versicherungen AG, Health Sciences, Zurich

HWZ University of Applied Sciences in Business Administration, Zurich

andri@signorell.net

August 1st, 2026

Overview

This vignette introduces combinatorial concepts and their implementation in the bedrock R package (from DescToolsX ecosystem). Let n denote the number of elements and m the sample size.

Overview of cases:

Permutations (all elements):	$n! \rightarrow \text{factorial}(n)$,	<code>permn()</code>
Ordered, no replacement:	$n! / (n-m)!$	<code>combN(n, m, ord=TRUE)</code>
Unordered, no replacement:	<code>choose(n, m)</code>	<code>choose()</code> , <code>combN()</code>
Ordered, with replacement:	$n^m \rightarrow n^m$	<code>combN(repl=TRUE, ord=TRUE)</code>
Unordered, with replacement:	<code>choose(n+m-1, m)</code>	<code>choose()</code> , <code>combN(repl=TRUE)</code>

0. Key concepts

- **Permutation:** arrangement in some order.
- **Ordered versus unordered samples:** In ordered samples, the order of the elements in the sample matters; e.g. digits in a phone number, or the letters in a word. In unordered samples the order of the elements is irrelevant; e.g. elements in a subset, fruits in a fruit salad or lottery numbers.
- **Samples with replacement versus samples without replacement:** In the first case, repetition of the same element is allowed (e.g. numbers in a license plate); in the second, repetition not allowed (as in a lottery drawing—once a number has been drawn, it cannot be drawn again).

This document demonstrates the calculations with an example of the letters “a”, “b”, “c” and “d”, and so $n = 4$. The formulas for the number of the permutations and combinations are given, as well as an approach how to construct the respective samples.

```
library(bedrock)
```

```
x <- letters[1:4]  
n <- length(x)  
m <- 2
```

Performance note

The size of combinatorial sets grows rapidly (e.g. $O(n!)$). Avoid generating full sets for $n > \sim 8$ unless necessary. Let fortune 220 inspire you:

```
fortunes::fortune(220)
```

Fabio Mulazzani: I need to obtain all the 9.somethingExp157 permutations that can be given from the numbers from 1 to 100.

Ted Harding: To an adequate approximation there are 10^{158} of them. Simply to obtain them all (at a rate of 10^{10} per second, which is faster than the CPU frequency of most desktop computers) would take 10^{148} seconds, or slightly longer than $3 \cdot (10^{140})$ years.

Current estimates of the age of the Universe are of the order of $1.5 \cdot (10^{10})$ years, so the Universe will have to last about $2 \cdot (10^{130})$ times as long as it has already existed, before the task could be finished.

So: why do you want to do this?

*-- Fabio Mulazzani and Ted Harding
R-help (November 2008)*

1. Some basic number functions

Note that Base R provides `factorial()` and `choose()`, but lacks a unified interface and tools to generate full sets. `bedrock` fills this gap.

`primes(x)` returns all prime numbers up to a certain limit x . `isPrime()` checks if a number x is a prime number.

`factorize()` splits a number in its prime bases and returns the number and the specifically used power.

`GCD()` returns the greatest common divisor and `LCM()` the least common multiple of a vector of numbers.

`divisors()` returns all the divisors of positive natural numbers.

```

# Find all prime numbers less than n
primes(n=20)                                # Package bedrock

## [1]  2  3  5  7 11 13 17 19

set.seed(23)
(y <- sample(1:20, 5))                      # base R

## [1] 12  5  6 13 14

# check if the elements in x are prime numbers
isPrime(y)                                  # Package bedrock

## [1] FALSE  TRUE FALSE  TRUE FALSE

# compute the prime factorizations of integers n
factorize(n=c(56, 42))                      # Package bedrock

## $`56`
##      p m
## [1,] 2 3
## [2,] 7 1
##
## $`42`
##      p m
## [1,] 2 1
## [2,] 3 1
## [3,] 7 1

# calculate divisors of positive natural numbers
divisors(c(221, 452))                      # Package bedrock

## $`221`
## [1]  1 13 17
## $`452`
## [1]  1  2  4 113 226

# calculate the greatest common divisor of the given numbers
GCD(64, 80, 160)                          # Package bedrock

## [1] 16

# calculate the least common multiple
LCM(10, 4, 12, 9, 2)                      # Package bedrock

## [1] 180

# calculate the first 12 Fibonacci numbers
fibonacci(1:12)                            # Package bedrock

## [1]  1  1  2  3  5  8 13 21 34 55 89 144

```

2. Permutations of n objects

The number of permutation of n objects is calculated as: $n!$

We can use the base function `factorial()` to calculate the total number of permutations. For creating the whole dataset of the permutations, there's the function `bedrock::permn()`.

```

factorial(n)                                # base R

## [1] 24

```

```
# generate all permutations
permn(x)
```

```
# Package bedrock
```

```
##      [,1] [,2] [,3] [,4]
## [1,] "a"  "b"  "c"  "d"
## [2,] "b"  "a"  "c"  "d"
## [3,] "b"  "c"  "a"  "d"
## [4,] "a"  "c"  "b"  "d"
## [5,] "c"  "a"  "b"  "d"
## [6,] "c"  "b"  "a"  "d"
## [7,] "b"  "c"  "d"  "a"
## [8,] "a"  "c"  "d"  "b"
## [9,] "c"  "a"  "d"  "b"
## [10,] "c"  "b"  "d"  "a"
## [11,] "a"  "b"  "d"  "c"
## [12,] "b"  "a"  "d"  "c"
```

```
## [13,] "c"  "d"  "a"  "b"
## [14,] "c"  "d"  "b"  "a"
## [15,] "a"  "d"  "b"  "c"
## [16,] "b"  "d"  "a"  "c"
## [17,] "b"  "d"  "c"  "a"
## [18,] "a"  "d"  "c"  "b"
## [19,] "d"  "a"  "b"  "c"
## [20,] "d"  "b"  "a"  "c"
## [21,] "d"  "b"  "c"  "a"
## [22,] "d"  "a"  "c"  "b"
## [23,] "d"  "c"  "a"  "b"
## [24,] "d"  "c"  "b"  "a"
```

3. Ordered samples of size m, without replacement, from n objects

The number is calculated as: $n \cdot (n-1) \dots (n-m+1) = \frac{n!}{(n-m)!} = {}_n P_r$

There's (to the author's knowledge) no function in R which would directly calculate this, but we can make use of the identity of the gamma function, which can be evaluated for any positive number n:

$$\Gamma(n) = (n-1)!$$

So ${}_n P_r$ can be calculated as $\text{gamma}(n+1)/\text{gamma}(n-m+1)$, respectively as $\exp(\text{lgamma}(n+1) - \text{lgamma}(n-m+1))$ to avoid numeric overflow. This is implemented so in `bedrock::combN()`.

A simpler (but computationally much more demanding) solution would be `prod((n-m+1):n)`.

For creating the whole dataset of the permutations, there's the function `bedrock::combSet()`, which has an argument `m` for defining the size of the subset to be drawn and where the replacement and order arguments can be set.

```
combN(n, m, repl=FALSE, ord=TRUE)
```

```
# Package bedrock
```

```
## [1] 12
```

```
# generate all samples
```

```
combSet(x, m, repl=FALSE, ord=TRUE)
```

```
# Package bedrock
```

```
##      [,1] [,2]
## [1,] "a"  "b"
## [2,] "b"  "a"
## [3,] "a"  "c"
## [4,] "c"  "a"
## [5,] "a"  "d"
## [6,] "d"  "a"
```

```
## [7,] "b"  "c"
## [8,] "c"  "b"
## [9,] "b"  "d"
## [10,] "d"  "b"
## [11,] "c"  "d"
## [12,] "d"  "c"
```

4. Unordered samples of size m, without replacement, from a set of n objects

The number is calculated as:
$$\binom{n}{m} = \frac{n P_r}{m!} = \frac{n \cdot (n-1) \dots (n-m+1)}{m!} = \frac{n!}{m! \cdot (n-m)!}$$

That's exactly what `choose()` returns.

For creating the whole dataset of the combinations, again the function `bedrock::combSet()` can be used.

```
choose(n, m) # base R
combN(n, m) # Package bedrock
## [1] 6
# generate all samples
combSet(x, m, repl=FALSE, ord=FALSE) # Package bedrock
##      [,1] [,2]
## [1,] "a"  "b"
## [2,] "a"  "c"
## [3,] "a"  "d"
## [4,] "b"  "c"
## [5,] "b"  "d"
## [6,] "c"  "d"
```

5. Ordered samples of size m, with replacement, from n objects

The number is calculated as: n^m

This can directly be entered as n^m into R.

For creating the whole dataset of the combinations, again the function `bedrock::combSet()` can be used.

This solution is based on Rs `expand.grid()` and `replicate()` functions.

```
n^m # base R
## [1] 16
# or as alternative
combN(n, m, repl=TRUE, ord=TRUE) # Package bedrock
## [1] 16
# generate all samples
combSet(x, m, repl=TRUE, ord=TRUE) # Package bedrock
##      [,1] [,2]
## [1,] "a"  "a"
## [2,] "b"  "a"
## [3,] "c"  "a"
## [4,] "d"  "a"
## [5,] "a"  "b"
## [6,] "b"  "b"
## [7,] "c"  "b"
## [8,] "d"  "b"
## [9,] "a"  "c"
## [10,] "b" "c"
## [11,] "c" "c"
## [12,] "d" "c"
## [13,] "a" "d"
## [14,] "b" "d"
## [15,] "c" "d"
## [16,] "d" "d"
```

6. Unordered samples of size m, with replacement, from a set of n objects

The number is calculated as:
$$\binom{n+m-1}{m} = \frac{(n+m-1)!}{m! \cdot (n-1)!}$$

Here again we can use the function `choose` with the appropriate arguments.

Creating the whole dataset of the combinations, needed the most sophisticated approach. The idea is to start with the dataset from the ordered samples of size m, without replacement (4.) and keep only the unique entries (concerning the containing letters). This is encapsulated in the function

`bedrock::combSet()`.

Again for the numbers either the base R solution with `choose` or the `bedrock::combN()` can be used.

```
choose(n + m - 1, m)                # base R

## [1] 10

# or as alternative
combN(n, m, repl=TRUE, ord=FALSE)    # Package bedrock

## [1] 10

# generate all samples
combSet(x, m, repl=TRUE, ord=FALSE)  # Package bedrock

##      [,1] [,2]
## [1,] "a"  "a"
## [2,] "a"  "b"
## [3,] "a"  "c"
## [4,] "a"  "d"
## [5,] "b"  "b"
## [6,] "b"  "c"
## [7,] "b"  "d"
## [8,] "c"  "c"
## [9,] "c"  "d"
## [10,] "d"  "d"
```

7. Generate all possible subsets

A list with all the subsets that can be built based on the elements given in x can be created by setting the argument m in `bedrock::combSet()` to a vector of all desired lengths. (For all subsets set m=1:4 in the following example)

```
combSet(letters[1:4], m=2:3)         # Package bedrock

## [[1]]
## [,1] [,2]
## [1,] "a" "b"
## [2,] "a" "c"
## [3,] "a" "d"
## [4,] "b" "c"
## [5,] "b" "d"
## [6,] "c" "d"
##
## [[2]]
## [,1] [,2] [,3]
## [1,] "a" "b" "c"
## [2,] "a" "b" "d"
## [3,] "a" "c" "d"
## [4,] "b" "c" "d"
```

8. Pairs from two different sets

If the pairs between two different sets should be found, there's the function `bedrock::combPairs()`.

```
combPairs(letters[1:3], letters[4:6])          # Package bedrock

##      Var1 Var2
## 1      a    d
## 2      b    d
## 3      c    d
## 4      a    e
## 5      b    e
## 6      c    e
## 7      a    f
## 8      b    f
## 9      c    f
```

9. Generating random groups

The function `bedrock::randGroupSplit()` can be used to produce random groups.

```
randGroupSplit(LETTERS[1:17], groupSizes = c(7,6,4))

## $`1`
## [1] "D" "E" "F" "J" "L" "O" "P"
##
## $`2`
## [1] "A" "B" "H" "K" "N" "Q"
##
## $`3`
## [1] "C" "G" "I" "M"
## [1] "A" "B" "C" "D" "E"
```

10. R-Code (condensed)

```
library(bedrock)
x <- letters[1:4]
n <- length(x)
m <- 2

# Find all prime numbers less than n
primes(n=20)
set.seed(23)
(y <- sample(1:20, 5))

# check if the elements in x are prime numbers
isPrime(y)
# compute the prime factorizations of integers n
factorize(n=c(56, 42))
# calculate divisors of positive natural numbers
divisors(c(221, 452))
# calculate the greatest common divisor of the
# given numbers
GCD(64, 80, 160)
# calculate the least common multiple
LCM(10, 4, 12, 9, 2)
# calculate the first 12 Fibonacci numbers
fibonacci(1:12)

# Permutations of n objects
factorial(n)
# generate all permutations
permn(x)

# Ordered samples of size m, without
# replacement, from n objects
combN(n, m, repl=FALSE, ord=TRUE)
# generate all samples
combSet(x, m, repl=FALSE, ord=TRUE)

# Unordered samples of size m, without
# replacement, from a set of n objects
choose(n, m)
combN(n, m)

# generate all samples
combSet(x, m, repl=FALSE, ord=FALSE)

# Ordered samples of size m, with replacement,
# from n objects
n^m
# or as alternative
combN(n, m, repl=TRUE, ord=TRUE)
# generate all samples
combSet(x, m, repl=TRUE, ord=TRUE)

# Unordered samples of size m, with
# replacement, from a set of n objects
choose(n + m - 1, m)
# or as alternative
combN(n, m, repl=TRUE, ord=FALSE)
# generate all samples
combSet(x, m, repl=TRUE, ord=FALSE)

# Generate all possible subsets
combSet(letters[1:4], m=2:3)

# Pairs from two different sets
combPairs(letters[1:3], letters[4:6])

# Generating random groups
(x <- LETTERS[1:5])
set.seed(78)
sample(x)
(grp <- sample(rep(c("A","B","C"),
                  c(4, 3, 5))))
sapply(c("A","B","C"), function(x)
which(grp==x))
```

11. References

Wollschläger, D. (2010, 2012) Grundlagen der Datenanalyse mit R, Springer, Berlin.