# R Generate Arrays

Fan Wang

2021-04-03

## Contents

## 1 Generate Arrays

Go to the **RMD**, **R**, **PDF**, or **HTML** version of this file. Go back to fan's REconTools research support package, R4Econ examples page, PkgTestR packaging guide, or Stat4Econ course page.

### 1.1 Generate Often Used Arrays

#### 1.1.1 Equi-distance Array with Bound

Consider multiple income groups in income bins that are equal-width, for the final income group, consider all individuals above some final bin minimum bound. Below the code generates this array of numbers: $0, 20000, 40000, 60000, 80000, 100000, 100000000$.

```r
# generate income cut-offs
fl_bin_start <- 0
# width equal to 20,000
fl_bin_width <- 2e4
# final point is 100 million
fl_bin_final_end <- 1e8
# final segment starting point is 100,000 dollars
fl_bin_final_start <- 1e5
# generate tincome bins
ar_income_bins <- c(
  seq(fl_bin_start, fl_bin_final_start, by = fl_bin_width),
  fl_bin_final_end
)
# Display
print(ar_income_bins)
```

```
## [1] 0e+00 2e+04 4e+04 6e+04 8e+04 1e+05 1e+08
```

Generate finer bins, at 5000 USD intervals, and stopping at 200 thousand dollars.

```r
fl_bin_start <- 0
fl_bin_width <- 5e3
fl_bin_final_end <- 1e8
fl_bin_final_start <- 2e5
ar_income_bins <- c(
  seq(fl_bin_start, fl_bin_final_start, by = fl_bin_width),
  fl_bin_final_end
)
print(ar_income_bins)
```

```
##  [1] 0.00e+00 5.00e+03 1.00e+04 1.50e+04 2.00e+04 2.50e+04 3.00e+04 3.50e+04 4.00e+04
## [10] 4.50e+04 5.00e+04 5.50e+04 6.00e+04 6.50e+04 7.00e+04 7.50e+04 8.00e+04 8.50e+04
## [19] 9.00e+04 9.50e+04 1.00e+05 1.05e+05 1.10e+05 1.15e+05 1.20e+05 1.25e+05 1.30e+05
## [28] 1.35e+05 1.40e+05 1.45e+05 1.50e+05 1.55e+05 1.60e+05 1.65e+05 1.70e+05 1.75e+05
## [37] 1.80e+05 1.85e+05 1.90e+05 1.95e+05 2.00e+05 1.00e+08
```

#### 1.1.2 Log Space Arrays

Often need to generate arrays on log rather than linear scale, below is log 10 scaled grid.

```r
# Parameters
it.lower.bd.inc.cnt <- 3
fl.log.lower <- -10
fl.log.higher <- -9
fl.min.rescale <- 0.01
it.log.count <- 4
# Generate
ar.fl.log.rescaled <- exp(log(10) * seq(log10(fl.min.rescale),
  log10(fl.min.rescale +
    (fl.log.higher - fl.log.lower)),
  length.out = it.log.count
))
ar.fl.log <- ar.fl.log.rescaled + fl.log.lower - fl.min.rescale
# Print
ar.fl.log
```

```
## [1] -10.000000  -9.963430  -9.793123  -9.000000
```

## 1.2 Generate Arrays Based on Existing Arrays

### 1.2.1 Probability Mass Array and Discrete Value Array

There are two arrays, an array of values, and an array of probabilities. The probability array sums to 1. The array of values, however, might not be unique.

First, generate some array of numbers not sorted and some proability mass for each non-sorted, non-unique element of the array.

```r
set.seed(123)
it_len <- 10
ar_x <- ceiling(runif(it_len) * 5 + 10)
ar_prob <- dbinom(seq(0, it_len - 1, length.out = it_len), it_len - 1, prob = 0.5)
print(cbind(ar_x, ar_prob))
```

```
##       ar_x     ar_prob
##  [1,]   12 0.001953125
##  [2,]   14 0.017578125
##  [3,]   13 0.070312500
##  [4,]   15 0.164062500
##  [5,]   15 0.246093750
##  [6,]   11 0.246093750
##  [7,]   13 0.164062500
##  [8,]   15 0.070312500
##  [9,]   13 0.017578125
## [10,]   13 0.001953125
```

```
print(paste0("sum(ar_prob)=", sum(ar_prob)))
```

```
## [1] "sum(ar_prob)=1"
```

Second, sorting index for ar_x, and resort ar_prob with the same index:

```
ls_sorted_res <- sort(ar_x, decreasing = FALSE, index.return = TRUE)
ar_idx_increasing_x <- ls_sorted_res$ix
ar_x_sorted <- ls_sorted_res$x
ar_prob_sorted <- ar_prob[ar_idx_increasing_x]
print(cbind(ar_x_sorted, ar_prob_sorted))
```

```
##       ar_x_sorted ar_prob_sorted
##  [1,]          11    0.246093750
##  [2,]          12    0.001953125
##  [3,]          13    0.070312500
##  [4,]          13    0.164062500
##  [5,]          13    0.017578125
##  [6,]          13    0.001953125
##  [7,]          14    0.017578125
##  [8,]          15    0.164062500
##  [9,]          15    0.246093750
## [10,]          15    0.070312500
```

Third, sum within group and generate unique, using the aggregate function. Then we have a column of unique values and associated probabilities.

```
ar_x_unique <- unique(ar_x_sorted)
mt_prob_unique <- aggregate(ar_prob_sorted, by = list(ar_x_sorted), FUN = sum)
ar_x_unique_prob <- mt_prob_unique$x
print(cbind(ar_x_unique, ar_x_unique_prob))
```

```
##      ar_x_unique ar_x_unique_prob
## [1,]          11      0.246093750
## [2,]          12      0.001953125
## [3,]          13      0.253906250
## [4,]          14      0.017578125
## [5,]          15      0.480468750
```

Finally, the several steps together.

```
# data
set.seed(123)
it_len <- 30
ar_x <- ceiling(runif(it_len) * 20 + 10)
ar_prob <- runif(it_len)
ar_prob <- ar_prob / sum(ar_prob)
# step 1, sort
```

```r
ls_sorted_res <- sort(ar_x, decreasing = FALSE, index.return = TRUE)
# step 2, unique sorted
ar_x_unique <- unique(ls_sorted_res$x)
# step 3, mass for each unique
mt_prob_unique <- aggregate(ar_prob[ls_sorted_res$ix], by = list(ls_sorted_res$x), FUN = sum)
ar_x_unique_prob <- mt_prob_unique$x
# results
print(cbind(ar_x_unique, ar_x_unique_prob))
```

```
##       ar_x_unique ar_x_unique_prob
##  [1,]          11      0.071718383
##  [2,]          13      0.040040920
##  [3,]          15      0.017708800
##  [4,]          16      0.141199002
##  [5,]          17      0.020211876
##  [6,]          19      0.052488290
##  [7,]          20      0.049104113
##  [8,]          21      0.067328518
##  [9,]          22      0.109454333
## [10,]          23      0.060712145
## [11,]          24      0.107671406
## [12,]          25      0.015694798
## [13,]          26      0.068567789
## [14,]          28      0.090925756
## [15,]          29      0.001870451
## [16,]          30      0.085303420
```

## 1.3 Generate Integer Sequences

### 1.3.1 Gapped Possibly Overlapping Consecutive Sequences

Now, we generate a set of integer sequences, with gaps in between, but possibly overlapping, for example: $(1, 2, 3, 4, 5), (5, 6), (10, 11)$.

First, we select a small random subset of integers between min and max, and we generate randomly a sequence of `length.out` of the same length. Each `length.out` up to a max. (we adjust in apply in the next block to make sure max given duration does not exceed bound.)

```r
# Number of random starting index
it_start_idx <- 11
it_end_idx <- 100
it_startdraws <- 6
# Maximum duration
it_duramax <- 3

# Random seed
set.seed(987)
# Draw random index between min and max
ar_it_start_idx <- sample(
  x = seq(from = it_start_idx, to = it_end_idx, by = 1),
  size = it_startdraws, replace = FALSE
)
ar_it_start_idx <- sort(ar_it_start_idx)
# Draw random durations, replace = TRUE because can repeat
ar_it_duration <- sample(
  x = it_duramax, size = it_startdraws, replace = TRUE
)

# Print
print(glue::glue(
  "random starts + duration: ",
  "{ar_it_start_idx} + {ar_it_duration}"
))
```

```
## random starts + duration: 35 + 3
## random starts + duration: 39 + 3
## random starts + duration: 42 + 1
## random starts + duration: 56 + 2
## random starts + duration: 57 + 1
## random starts + duration: 73 + 1
```

Second, we expand the indexes with neighboring values, and create a list of consecutive integer sequences.

```r
# start and end sequences
# note the min operator inside, the makes sure we do not exceed max
ls_ar_it_recession <- apply(
  cbind(ar_it_start_idx, ar_it_start_idx + ar_it_duration),
  1, function(row) {
    return(seq(row[1], min(row[2], it_end_idx)))
  }
)
# Draw it_m from indexed list of it_N
print("ls_ar_it_recession")
```

```
## [1] "ls_ar_it_recession"
```

```r
print(ls_ar_it_recession)
```

```
## [[1]]
## [1] 35 36 37 38
##
## [[2]]
## [1] 39 40 41 42
##
## [[3]]
## [1] 42 43
##
## [[4]]
## [1] 56 57 58
##
## [[5]]
## [1] 57 58
##
## [[6]]
## [1] 73 74
```

Third, we can bring the sequences generated together if we want to

```r
# Combine arrays
ar_it_recession_year <- (
  sort(do.call(c, ls_ar_it_recession))
)
```

```r
# Print
print(glue::glue(
  "print full as array:",
  "{ar_it_recession_year}"
))
```

```
## print full as array:35
## print full as array:36
## print full as array:37
## print full as array:38
## print full as array:39
## print full as array:40
## print full as array:41
## print full as array:42
## print full as array:42
## print full as array:43
## print full as array:56
## print full as array:57
## print full as array:57
## print full as array:58
## print full as array:58
## print full as array:73
## print full as array:74
```

### 1.3.2 Gapped non-Overlapping Consecutive Sequences

Now, we generate a set of integer sequences, with gaps in between, but not overlapping, for example: $(1, 2, 3), (5, 6), (10, 11)$. We follow a very similar structure as above, but now adjust starting draws by prior accumulated durations.

Note that in the code below, we could end up with less that `it_startdraws` if there are consecutive start draws. We can only have non-consecutive start draws to avoid overlaps.

```r
# Number of random starting index
it_start_idx <- 11
it_end_idx <- 100
it_startdraws_max <- 6
it_duramax <- 3

# Random seed
set.seed(987)
# Draw random index between min and max
ar_it_start_idx <- sort(sample(
  seq(it_start_idx, it_end_idx),
  it_startdraws_max,
  replace = FALSE
))
# Draw random durations, replace = TRUE because can repeat
ar_it_duration <- sample(it_duramax, it_startdraws_max, replace = TRUE)

# Check space between starts
ar_it_startgap <- diff(ar_it_start_idx)
ar_it_dura_lenm1 <- ar_it_duration[1:(length(ar_it_duration) - 1)]
# Adjust durations
ar_it_dura_bd <- pmin(ar_it_startgap - 2, ar_it_dura_lenm1)
```

```r
ar_it_duration[1:(length(ar_it_duration) - 1)] <- ar_it_dura_bd

# Drop consecutive starts
ar_bl_dura_nonneg <- which(ar_it_duration >= 0)
ar_it_start_idx <- ar_it_start_idx[ar_bl_dura_nonneg]
ar_it_duration <- ar_it_duration[ar_bl_dura_nonneg]

# list of recession periods
ls_ar_it_recession_non_overlap <- apply(
  cbind(ar_it_start_idx, ar_it_start_idx + ar_it_duration),
  1, function(row) {
    return(seq(row[1], min(row[2], it_end_idx)))
  }
)

# print
print("ls_ar_it_recession_non_overlap")
```

```
## [1] "ls_ar_it_recession_non_overlap"
```

```r
print(ls_ar_it_recession_non_overlap)
```

```
## [[1]]
## [1] 35 36 37
##
## [[2]]
## [1] 39 40
##
## [[3]]
## [1] 42 43
##
## [[4]]
## [1] 57 58
##
## [[5]]
## [1] 73 74
```