About half a year ago I wrote about [dqsample](https://blog.daqana.com/en/dqsample-a-bias-free-alternative-to-rs-sample-function/) providing a bias free alternative to base::sample(). Meanwhile this is mostly of historic interest, since the upcomming R release 3.6.0 will see an [updated sampling algorithm](https://bugs.r-project.org/bugzilla/show_bug.cgi?id=17494). However, some of the techniques used in dqsample are now part of [dqrng](https://www.daqana.org/dqrng/). Of course, the sampling method is still free of the previously described bias:

m <- 2/5 \* 2^32

x <- dqrng::dqsample(floor(m), 1000000, replace = TRUE)

plot(density(x[x %% 2 == 0]), main = "dqrng::dqsample", xlab = NA)

lines(density(x[x %% 2 == 1]), col = "#FF8F00")

This most recent version of dqrng is not on CRAN yet, but you can install it via [drat](https://cran.r-project.org/package=drat):

if (!requireNamespace("drat", quietly = TRUE)) install.packages("drat")

drat::addRepo("daqana")

install.packages("dqrng")

Overall we will see that it is possible to improve on R’s sampling speed without compromising statistical qualities. [Feedback](https://github.com/daqana/dqrng/issues) is always welcome!

**Benchmarks**

The following benchmarks were made with R version 3.5.3 (2019-03-11) running under Debian GNU/Linux 9 (stretch) on a Intel® Core i7-6600U CPU @ 2.60GHz. YMMV, as always with benchmarks …

By combining fast RNGs with fast methods for creating [integers in a range](http://www.pcg-random.org/posts/bounded-rands.html) one gets good performance for sampling with replacement:

library(dqrng)

m <- 1e6

n <- 1e4

bm <- bench::mark(sample.int(m, n, replace = TRUE),

sample.int(1e4\*m, n, replace = TRUE),

dqsample.int(m, n, replace = TRUE),

dqsample.int(1e4\*m, n, replace = TRUE),

check = FALSE)

knitr::kable(bm[, 1:6])

| **expression** | **min** | **mean** | **median** | **max** | **itr/sec** |
| --- | --- | --- | --- | --- | --- |
| sample.int(m, n, replace = TRUE) | 94.9µs | 112.5µs | 112.2µs | 3.78ms | 8886.863 |
| sample.int(10000 \* m, n, replace = TRUE) | 170.9µs | 202.8µs | 200.9µs | 3.08ms | 4930.094 |
| dqsample.int(m, n, replace = TRUE) | 31.5µs | 36.2µs | 35.7µs | 183.63µs | 27596.508 |
| dqsample.int(10000 \* m, n, replace = TRUE) | 35.7µs | 41.3µs | 39.8µs | 3.06ms | 24186.338 |

plot(bm)

Note that sampling from 10^10 integers triggers “[long-vector support](https://stat.ethz.ch/R-manual/R-devel/library/base/html/LongVectors.html)” in R.

When sampling *without* replacement one has to consider an appropriate algorithm for making sure that no entry is repeated. When more than 50% of the population are sampled, dqrng shuffles an appropriate part of the full list and returns that. The algorithm used in R is similar but dqrng has the edge with respect to performance:

library(dqrng)

m <- 1e6

n <- 6e5

bm <- bench::mark(sample.int(m, n),

dqsample.int(m, n),

check = FALSE, min\_iterations = 50)

knitr::kable(bm[, 1:6])

| **expression** | **min** | **mean** | **median** | **max** | **itr/sec** |
| --- | --- | --- | --- | --- | --- |
| sample.int(m, n) | 11.91ms | 14.09ms | 13.02ms | 26.11ms | 70.98544 |
| dqsample.int(m, n) | 6.93ms | 8.22ms | 8.17ms | 9.99ms | 121.65317 |

plot(bm)

For lower sampling ratios a set based rejection sampling algorithm is used by dqrng. In principle, R can make use of a similar algorithm based on a hashset. However, it is only used for larger input vectors even though it is faster than the default method. The algorithm in dqrng, which is based on a [bitset](https://lemire.me/blog/2012/11/13/fast-sets-of-integers/), is even faster, though:

library(dqrng)

m <- 1e6

n <- 1e4

bm <- bench::mark(sample.int(m, n),

sample.int(m, n, useHash = TRUE),

dqsample.int(m, n),

check = FALSE)

knitr::kable(bm[, 1:6])

| **expression** | **min** | **mean** | **median** | **max** | **itr/sec** |
| --- | --- | --- | --- | --- | --- |
| sample.int(m, n) | 788µs | 1.72ms | 1.87ms | 5.01ms | 583.0849 |
| sample.int(m, n, useHash = TRUE) | 229µs | 269.6µs | 266.15µs | 453.71µs | 3709.2196 |
| dqsample.int(m, n) | 113µs | 130.4µs | 129.49µs | 2.98ms | 7668.6069 |

plot(bm)

As one decreases the sampling rate even more, dqrng switches to a hashset based rejection sampling. Both hashset based methods have similar performance and are much faster than R’s default method.

library(dqrng)

m <- 1e6

n <- 1e2

bm <- bench::mark(sample.int(m, n),

sample.int(m, n, useHash = TRUE),

dqsample.int(m, n),

check = FALSE)

knitr::kable(bm[, 1:6])

| **expression** | **min** | **mean** | **median** | **max** | **itr/sec** |
| --- | --- | --- | --- | --- | --- |
| sample.int(m, n) | 452.72µs | 1.29ms | 1.56ms | 4.69ms | 777.0096 |
| sample.int(m, n, useHash = TRUE) | 3.98µs | 5.38µs | 5.03µs | 62.94µs | 185978.0662 |
| dqsample.int(m, n) | 3.59µs | 4.14µs | 4.05µs | 26.85µs | 241454.9528 |

plot(bm)

For larger sampling ranges R uses the hashset by default, though dqsample.int is still faster:

library(dqrng)

m <- 1e10

n <- 1e5

bm <- bench::mark(sample.int(m, n),

dqsample.int(m, n),

check = FALSE)

knitr::kable(bm[, 1:6])

| **expression** | **min** | **mean** | **median** | **max** | **itr/sec** |
| --- | --- | --- | --- | --- | --- |
| sample.int(m, n) | 5.3ms | 5.79ms | 5.48ms | 9.5ms | 172.7915 |
| dqsample.int(m, n) | 1.7ms | 2.04ms | 1.97ms | 5.08ms | 490.8562 |

plot(bm)

**Details**

The following methods are used for sampling without replacement. The algorithms are presented in R-like pseudocode, even though the [real implementation](https://github.com/daqana/dqrng/blob/95ca935218d5fe191dce99d318db4b3007b3d886/src/dqrng.cpp#L111-L140) is in C++. For sampling rates above 50%, a partial [Fisher-Yates shuffle](https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle) is used:

no\_replace\_shuffle <- function(m, n) {

tmp <- seq\_len(m)

for (i in seq\_len(n))

swap(tmp[i], tmp[i + random\_int(m-i)])

tmp[1:n]

}

where random\_int(m-i) returns a random integer in [0, m-i]. Since the full population is kept in memory, this method is only suitable for high selection rates. One could expect that [reservoir sampling](https://en.wikipedia.org/wiki/Reservoir_sampling) should work well for lower selection rates. However, in my tests set based algorithms were faster:

no\_replace\_set <- function(m, n) {

result <- vector(mode = "...", length = n) # integer or numeric

elems <- new(set, m, n) # set object for storing n objects out of m possible values

for (i in seq\_len(n))

while (TRUE) {

v = random\_int(m)

if (elems.insert(v)) {

result[i] = v

break

}

}

result

}

Here elems.insert(v) returns TRUE if the insert was successful, i.e. v was not in elems before, and FALSE otherwise. There are different strategies for implementing such a set. For intermediate sampling rates (currently between 0.1% and 50%) dqrng uses a bitset, i.e. a vector of m bits each representing one of the possible values. For lower sampling rates the memory usage of this algorithm is to expensive, which is why a hashset1 is used, since there the used memory scales with n and not with m. One could expect that [Robert Floyd’s sampling algorithm](https://stackoverflow.com/a/2394292/8416610) would be superior, but this was not the case in my tests, probably because it requires a final shuffling of the result to get a random *permutation* instead of a random *combination*.

1. For the specialists: Open addressing with a power-of-two size between 1.5 and 3 times n, identity hash function for the stored integers and quadratic probing.