Sequential Random Sampling Revisited: Hidden Shuffle Method

Michael Shekelyan
University of Warwick

Graham Cormode
University of Warwick

Abstract

Random sampling (without replacement) is ubiquitously employed to obtain a representative subset of the data. Unlike common methods, sequential methods report samples in ascending order of index without keeping track of previous samples. This enables lightweight iterators that can jump directly from one sampled position to the next. Previously, sequential methods focused on drawing from the distribution of gap sizes, which requires intricate algorithms that are difficult to validate and can be slow in the worst-case. This can be avoided by a new method, the Hidden Shuffle. The name mirrors the fact that although the algorithm does not resemble shuffling, its correctness can be proven by conceptualising the sampling process as a random shuffle. The Hidden Shuffle algorithm stores just a handful of values, can be implemented in few lines of code, offers strong worst-case guarantees and is shown to be faster than state-of-the-art methods while using comparably few random variates.

1 Introduction

Drawing a uniform random sample of items is a procedure that is foundational to a plethora of statistical and computational tasks, from hypothesis testing to scalable machine learning. Viewed as a randomized computational primitive, uniform sampling is simple to state, and widely implemented in programming languages, libraries and tools. It is so fundamental that little attention is typically paid to the process of drawing a sample. However, when we start to consider the constraints that arise when working with large volumes of data, it becomes apparent that sampling is not so trivial as it first seems.

The challenge arises when we do not have full random access to the (discrete) population from which we wish to draw the sample. For example, this may be when the items are being observed incrementally as a stream of observations, or when the items are distributed over a collection of observers. These scenarios have been addressed by Reservoir Sampling (Vitter, 1985; Efraimidis and Spirakis, 2006), and distributed sampling (Cormode et al., 2010; Tirthapura and Woodruff, 2011). In this paper, we revisit the question of *Sequential Random Sampling* (SRS). Here, the size of the population N is fixed, but we must emit the n sampled items in the order in which they are stored. The original motivation for SRS in the 1980s was data stored on tape, such that a single linear pass over the input was feasible while random access or multiple passes were not. The additional requirement of SRS is that each sampled item should be emitted as soon as it is observed, for immediate processing, so that the sampling algorithm should have a limited memory footprint.

In the intervening years, the importance of tape storage has diminished. However, the question of SRS remains fundamental, and new motivations have arisen due to the ballooning of data set sizes on distributed storage or received as data streams. Here, the prohibition of random access, and the requirement to emit samples as they are observed re-emerges, due to large item sizes and large sample sizes.

1.1 Baseline Sampling Methods

More formally, we want to obtain a simple random sample (without replacement) of size n from a population of size N . This means that any of the possible subsets has to be equally likely to be selected. For any combination of k items, the probability that they all appear in the simple random sample is equal to the number of subsets containing the combination $\binom{N-k}{n-k}$ divided by the total number of subsets $\binom{N}{n}$, which is equal to $\prod_{j=0}^{k-1} \frac{n-j}{N-j}$. The inclusion probability for an item is therefore $\frac{n}{N}$ and for a pair of items $\frac{n(n-1)}{N(N-1)}$.

Proceedings of the 24[th] International Conference on Artificial Intelligence and Statistics (AISTATS) 2021, San Diego, California, USA. PMLR: Volume 130.

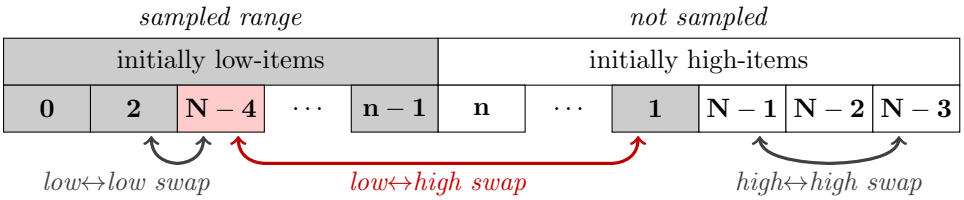

Figure 1: Conceptualising sampling as a shuffle for sample size n and population size N with example swaps.

A common algorithm used in libraries[1] is to obtain a sample by independently generating positions between 0 and $N-1$ and use a hash-map to detect duplicates. This can work well when the sample size n is very small. A downside of such an approach is that the hash-map can grow up to n entries and we would need to extract the sampled locations in sorted order if we wish to visit the sampled items sequentially.

A simple approach that addresses these two problems is to go through all items in ascending order and select each item $i \in \{0, \ldots, N-1\}$ with probability equal to the number of items left to sample divided by the number of remaining items $N - i$ as in Jones (1962). Here, the time cost is proportional to N, and we need to make $O(N)$ random sampling decisions. This is acceptable when n is close to N, but otherwise we seek solutions whose cost is closer to n. This was the focus of much of the early work on sequential random sampling, which sought to determine how many items to "jump over" before the next item to sample. However, it turns out that this distribution of gaps between items is not easy to deal with and leads to rather intricate algorithms such as Vitter's Algorithm D (Vitter, 1984, 1987) based on Neumann's rejection sampling. While the "difficulty" of algorithms is an elusive notion and may be too subjective to quantify, we nevertheless seek approaches that are "simple" to describe. Simplicity is highly desirable, not just for expository and didactic purposes, but because a simple algorithm can be (correctly) ported easily to all types of programming languages, libraries and frameworks.

1.2 Our Approach

In order to obtain a conceptually simpler algorithm, we break away from the perspective of directly accessing the distribution of gaps. Instead, we make use of a different paradigm, based on random permutations. We will consider the process that draws a sample by taking the first n items of a randomly permuted population of size N. Clearly, each subsequence of a randomly permuted sequence has to be a random subset, see also Sunter (1977). We can then simulate the execution of a random shuffle algorithm over the population items and keep track of which elements would end up as the first n items.

A trivial algorithm to randomly permute a set is to uniformly select a random element, remove it from the set and continue the same way with the remaining elements. The order in which items are removed forms a random permutation. An iterative formulation of this algorithm is Knuth's shuffle which steps through all positions between 0 and $N-1$ of an initial arbitrary ordering of the set, and swaps the item at position i with a random position between i and $N-1$.

We now argue that for a sample of size n, we can terminate this procedure early. Since we always select the first n items of a randomly permuted sequence, it is useful to conceptually differentiate between *low-items* whose position is smaller than n and the remaining *high-items* (cf. Figure 1). Then we can classify the swaps in Knuth's shuffle into low↔low, low↔high and high↔high swaps, where an X↔Y swap occurs when processing an X-item which is swapped with a Y-item. Observe that the algorithm makes no high↔low swaps, as at position i it cannot swap with items at a lower position. When Knuth's shuffle reaches position $i \geq n$, it can only perform high↔high swaps which do not influence the composition of our sample. Thus, we can focus our attention to low↔low and low↔high swaps.

This means we can just perform the first n swaps to obtain a sample. A similar idea is proposed by Devroye (1986) and later Bui (2015), but implementing this directly requires space proportional to n to record which items have been swapped into high positions. Moreover, to go through the sample sequentially, we need to collect and sort the full sample.

We can remove both of these limitations by realizing that we can postpone the selection of *which* items are swapped, and instead we just need to fix the *number* of sampled items in each category (low or high). Consequently, we sample the high-items and low-items separately. Once we determine how many high-items are in our sample, we immediately know the number of low-items. Sampling these low-items is easy to do, as the total number of low-items is small.

[1] The popular languages Python (`www.python.org`) and R (`www.r-project.org`) use it in their standard library.

Importantly, if we are only interested in the high-items in our sample, only low↔high swaps in Knuth's shuffle are relevant, as low↔low swaps only permute low-items. As the low↔high swaps only occur in the first n iterations of Knuth's shuffle, we can simulate the shuffle up to that point and count the number of low↔high swaps H. Since any high-item that is swapped into the sample can no longer be swapped out of the sample, we only need to determine the high-items of the low↔high swaps. Lastly, as each low↔high swap can select any high-item, the high-items of the low↔high swaps are just H independently drawn high-items.

As a result, we first draw the number of low↔high swaps H by simulating the shuffle up to position n, then draw H independent high-items, calculate the number of low-items L as n minus the number of distinct high-items and at the end draw L low-items. The key here is that the shuffling approach means that the draws to the high-items are *with replacement*, which is much easier to achieve. If we pick the same high location more than once, the semantics of shuffling are that the repetitions are interpreted as draws from the low region (since, in the shuffling algorithm, the first selection of a high-item would have swapped a low-item into its place). Hence, we just need to treat the duplicate samples as incrementing L, the count of low-items. The final step to sample L low-items is easy to do, as the total number of low-items is at most n in which case naive SRS algorithms such as Algorithm A of Vitter (1984) are sufficiently efficient for this final "mopping up" stage.

Now, the only open question is how do we draw the sample in ascending order. One trick we use is to obtain sampled indices in *descending* order but then mirror the sampled positions to obtain ascending order. As we can do this when we report samples, we can focus on sampling in descending order without loss of generality. It remains to get the H independent high-item locations (with replacement) in non-increasing order. This is a well-understood problem that can be answered efficiently. The basic idea is to draw uniform variates in descending order by exploiting order statistics and then employ inverse transform sampling. This uses the fact that the maximum of k standard uniform variates has a closed-form distribution given by $\sqrt[k]{U(0,1)}$. After we obtain the high-items, we can sample the low-items in descending order by employing Algorithm A (Vitter, 1984).

2 Related Work

Reservoir sampling (Vitter, 1985) is one of the best known sampling approaches as it requires no knowledge of the population size and maintains at any point a reservoir of items that is a simple random sample. One can efficiently jump to the next reservoir item (Li, 1994; Efraimidis and Spirakis, 2006), but in order to apply this for sequential sampling, one would again need to somehow identify which reservoir items will remain until the end or need to retain and sort the reservoir at the end. If the population size is known there exist many alternative methods to directly obtain sampled items.

In case of random access, the most common method is "selection sampling" where items are drawn independently while duplicates are rejected. The redundant draws can be avoided either by employing the swapping method (Devroye, 1986; Bui, 2015) or taking Robert Floyd's very simple and elegant approach (Bentley and Floyd, 1987). In case of sequential access, the most common method is to pick each item with an increasing probability, which confusingly is also often called "selection sampling" or Algorithm S (Fan et al., 1962; Jones, 1962; Bebbington, 1975; Knuth, 1969). As Algorithm S is not useful for very large population sizes, more sophisticated methods are discussed in the following.

Bernoulli Sampling picks each item with a fixed probability p and will also produce a simple random sample, but the size of the sample is no longer fixed and instead follows a binomial distribution with expected value pn. Such a sample can be obtained sequentially by generating geometric variates (that can be easily obtained from uniform variates) to draw gaps between items. Bernoulli Sampling can be adapted to give a fixed sample size by dropping random samples when oversampling and sampling multiple times when undersampling, e.g., Algorithm SG (Ahrens and Dieter, 1985), but this leads to delays in the reporting of samples and introduces a storage and computation overhead. The storage overhead can be reduced in case of pseudorandom number generation, where one can replay a sample by re-using the same seed (Ahrens and Dieter, 1985).

Hypergeometric distributions describe the number of items between subgroups (it can be thought of as a without-replacement analog of the binomial distribution) and can enable sequential random sampling (Sanders et al., 2018). The basic idea is to split the population items into two halves (for odd sizes picking a larger first half) and draw from the hypergeometric distribution with parameters $(N, \lceil \frac{N}{2} \rceil, n)$ to generate the number of samples in the first and second half and then recursively apply the same approach to both halves. By proceeding in a depth-first manner and skipping empty halves, this allows to obtain samples sequentially by generating $O(n)$ hypergeometric variates in total and storing up to $O(\log N)$ values, i.e.,

Table 1: Methods to obtain without-replacement sample of size n from population of size N.

	time	space	variates	latency	simplicity (subj.)
Conventional and Reservoir Sampling					
With-Replacement+Duplicate-Rejection	$\Theta_a(n)$	$\Theta(n)$	$\Theta_a(n)$	high	very simple
Swapping Method (Devroye, 1986)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	high	very simple
Floyd's [Alg. F] (Bentley and Floyd, 1987)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	high	very simple
Reservoir [Alg. R] (Vitter, 1985)	$\Theta(N)$	$\Theta(n)$	$\Theta(N)$	high	very simple
Reservoir-Skip [Alg. K,L,M,Z] (Li, 1994)	$\Theta_a(n \log \frac{N}{n})$	$\Theta(n)$	$\Theta_a(n \log \frac{N}{n})$	high	simple
Rand. Key (Efraimidis and Spirakis, 2006)	$\Theta_a(n \log \frac{N}{n})$	$\Theta(n)$	$\Theta_a(n \log \frac{N}{n})$	high	simple
Sequential Sampling (known population size)					
Sorted Non-Sequential	$\Omega_a(n \log n)$	$\Theta(n)$	$\Omega_a(n)$	high	very simple
Algorithm S (Fan et al., 1962; Jones, 1962)	$\Theta(N)$	$\Theta(1)$	$\Theta(N)$	high	very simple
Bernoulli (Ahrens and Dieter, 1985)	$\Theta_a(n)$	$\Theta(n)$	$\Theta(n)$	high	simple but tedious
Inverse-Transform [Alg. A] (Vitter, 1984)	$\Theta(N)$	$\Theta(1)$	$\Theta(n)$	high	simple
Hypergeometric (Sanders et al., 2018)	$\Omega(n)$	$\Omega_a(\log N)$	$\Omega(n)$	low	intricate variate gen.
Reject-Accept [Alg. D] (Vitter, 1984)	$\Theta_a(n)$	$\Theta(1)$	$\Theta_a(n)$	very low	complicated
Hidden Shuffle [Proposed]	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	low	simple

how many sampled items occur in the second halves at each level. A downside of this approach is that it shifts all of the complexity on to the generation of hypergeometric variates (Stadlober, 1990), which comes with many practical challenges especially when N is large[2].

Vitter (1984; 1987) and Devroye (1986) describe multiple methods to draw from the distribution of gaps between subsequent sampled items (denoted by $F(s)$), most notably Algorithms A and D. Algorithm A employs inverse transform sampling in combination with a sequential search through the CDF, i.e., it searches for the minimal s whose $F(s)$ is larger than a generated uniform variate, which requires at most $O(N)$ time as each $F(s+1)$ can be computed from $F(s)$ in $O(1)$. Algorithm D draws from a simpler distribution similar to the gap distribution and then rejects with an adequate probability to obtain the correct probabilities for each gap. If n is close to N the rejection probability is very large and the algorithm has to use Algorithm A as a fallback. The rejection framework of Algorithm D leads to a rather intricate algorithm, which so far has only been adopted to the standard library of the programming language `D`, despite good runtime performance and full support for sequential sampling. We show experimentally that our much simpler proposed approach is faster.

A comparison of approaches can be found in Table 1, including our (subjective) judgment on the simplicity of the algorithms. Here, asymptotic costs with a subscript 'a' denote average case, otherwise they show the worst case. We do not provide the formal complexities of latencies as none of the approaches achieves $O(1)$, and the formulas are more difficult to derive and interpret.

[2]The popular Python library `numpy` that provides an implementation of (Stadlober, 1990) cautions users to not exceed sizes of a billion and the algorithm requires the value of $\ln(k!)$ for $0 \leq k \leq N$ which has to be pre-computed for small values and approximated for large values.

3 Sequential Random Sampling

We now give the full description of our shuffling-based sequential random sampling (Hidden Shuffle) method. There exist efficient and straightforward sequential approaches for simple random sampling *without replacement* (WOR) from small populations, and for simple random sampling *with replacement* (WR) for any population size. The basic idea of our approach is to use the random shuffle as a conceptual model to reduce sequential WOR to these two simpler problems.

3.1 Shuffle-Based Sampler

Knuth's shuffle (Knuth, 1969) is a well-known algorithm to obtain a random permutation of a sequence:

Algorithm 1: Random shuffle (Knuth, 1969)

foreach $i \in (0, 1, \ldots, N-2)$ **do**
  $j \sim \mathcal{U}\{i, i+1, \ldots, N-1\}$
  swap items at positions i and j

In each iteration, Knuth's shuffle picks a random remaining element into the i-th position, such that after each iteration all preceding elements are a random subset of the population. Thus, if we terminate after the first n swaps, we obtain a WOR sample of size n. Conceptually, we can therefore consider running the first n iterations of Algorithm 1.

To describe our approach, we split the range of item indices $\{0, 1, \ldots, N\}$ into *low-items* $\{0, 1, \ldots, \ell - 1\}$ and *high-items* $\{\ell, \ell + 1, \ldots, N - 1\}$. In what follows, we assume that the split point ℓ is chosen as the target size of the sample, $\ell = n$. Accordingly, we can also view our sample as the union of various low-items and high-items, i.e., $\mathcal{S} = \mathcal{L} \cup \mathcal{H}$ where $\mathcal{L} \subseteq \{0, 1, \ldots, n-1\}$ and $\mathcal{H} \subseteq \{n, n+1, \ldots, N-1\}$. Our strategy is to first obtain the sample over the high-items $\mathcal{H}$ and then obtain the sample over the low-items $\mathcal{L}$ separately.

As the sampling is symmetrical, any algorithm that sequentially obtains samples in descending order can be converted to ascending order by reporting each sampled item x as $(N-1) - x$. We make use of this mirroring trick, as it is more convenient for us to sample items initially in descending order.

Algorithm 2: Hidden Shuffle Sampling Method

input : Population size N and sample size n
$H \sim n - \sum_{i=0}^{n-1} \text{Bernoulli}(1 - \frac{N-n}{N-i})$ ①
$\mathcal{H} \sim \text{Unique}(\text{WR}^{H}\{n, n+1, \ldots, N-1\})$ ②
$\mathcal{L} \sim \text{WOR}^{(n-|\mathcal{H}|)}\{0, 1, \ldots, n-1\}$ ③
output: random sample $(\mathcal{H} \cup \mathcal{L}) \subseteq \{0, \ldots, N-1\}$

Algorithm 2 gives the overview of our approach. In step ① we determine H, which counts how often Knuth's shuffle would swap position $i < n$ with a position $j \geq n$ (number of low↔high swaps, cf. Figure 1). As the probability that for all other cases is $p_i = 1 - \frac{N-n}{N-i}$, we can express it as a Poisson Binomial Distribution, i.e., $H \sim n - \sum_{i=0}^{n-1} B_i$ with $B_i \sim \text{Bernoulli}(p_i)$. We subtract here from n and model successes as not being a low↔high swap, because low↔high swaps are much more prevalent for large N and it simplifies the presentation to have small success probabilities in the discussion of step 2.

In step ② we (independently) sample H high-items with-replacement, which corresponds to the j's with which Knuth's shuffle will swap position $i < n$ with a position $j \geq n$. As this sampling is with-replacement, each j can be selected multiple times. The first time j is selected, Knuth's shuffle encounters a high-item at position j and swaps it with a low-item at position i. As a result, if j is selected again, Knuth's shuffle will now encounter a low-item at position j. Thus, repeated draws of j simply lead to more low-items and we just need to find the unique items to obtain the sampled high-items $\mathcal{H}$, which in turn determines the number of sampled low-items $|\mathcal{L}| = n - |\mathcal{H}|$. As we are sampling indices with-replacement here, we can reduce the sampling of $\mathcal{H}$ in descending order to sampling order statistics of standard uniform variates. Sampling in descending order also has the side effect that we can immediately detect repetitions of sampled items.

In step ③ we know the number of low-items $L = \mathcal{L}$ that we need to sample from the set of low-items. This is a (smaller) instance of the problem we started with, where the domain has shrunk from N to n. As the domain size is small, we can employ an approach such as Algorithm A that requires just one variate per sample. Alternatively, we could apply our proposed Hidden Shuffle approach recursively based on a smaller threshold ℓ until the sample is full.

3.2 Efficient Algorithm

Next, we describe how to efficiently implement the steps of the above outline approach.

The Full Algorithm. Code 1 shows our approach implemented in Python[3] (version 2.3 and upwards). It takes the population size `N` and sample size `n` as inputs, and outputs an iterator over the sampled items.

Step 1: Two-lane generation of Poisson Binomial Variates. In this step the number of low↔high swaps (cf. Code 1, lines 4-13) is generated, which is then used in the subsequent step to sample high-items with-replacement where duplicates are ignored and simply increase the number of sampled low-items.

The quantity H follows a Poisson binomial distribution for which Le Cam's theorem gives approximation bounds based on the Poisson distribution, but we instead aim to generate a variate efficiently with the help of the geometric distribution. The basic idea is that most of our Bernoulli trials are expected to fail and if they fail for a larger success probability q, then they will also fail for the correct one. The trick here is that q is shared between all subsequent trials. Thus, we can establish two lanes, an *express lane* for trials that fail even for a larger success probability q and a *slow lane* for trials that succeed for q, but might otherwise fail. We can then use the variates from the geometric distribution on the express lane and briefly switch to the slow lane when we encounter a failed trial. It follows a formal definition of this idea:

Theorem 3.1 (Generating Poisson Binomial)**.** *A variate $X = \sum_{i=1}^{n} B_i$ with $B_i \sim \text{Bernoulli}(p_i)$ can be drawn using $\lceil n \cdot \max_{i=1}^{n} p_i \rceil$ uniform variates on expectation.*

[3]The function `float` casts numbers to floating-point numbers, `int` rounds down, `uniform(0,1)` uniformly draws a random value between 0 and 1, `a**b` performs a^b and `yield` reports a result. In order to print all samples for $N = 10^9$ and $n = 10000$ using Code 1 one would call `for x in seqsample(10**9,10000): print(x);`

Code 1 Hidden Shuffle implemented in Python

```
 1: from math import log,exp,floor
 2: from random import uniform
 3: def seqsample(N, n): # WOR from 0..N-1
 4:     H = 0; i = 0 # STEP 1: compute H
 5:     if N > n:
 6:         H = n
 7:         while i < n:
 8:             q = 1.0-float(N-n)/(N-i)
 9:             i = i+int(log(uniform(0,1), 1-q) )
10:             p_i = 1.0-float(N-n)/max(N-i,1)
11:             if i < n and uniform(0,1) < p_i/q:
12:                 H = H-1
13:             i = i+1
14:     L = n-H; a = 1.0
15:     while H > 0: # STEP 2: draw high-items
16:         S_old = n+int(a*(N-n) )
17:         a = a * uniform(0,1)**(1.0/H)
18:         S = n+int(a*(N-n) )
19:         if S < S_old:
20:             yield (N-1) - S
21:         else:
22:             L = L+1  # duplicate detected
23:         H = H-1
24:     while L > 0: # STEP 3: draw low-items
25:         u = uniform(0,1); s=0; F=float(L)/n
26:         while F < u and s < (n-L):
27:             F = 1-(1-float(L)/(n-s))*(1-F)
28:             s = s+1
29:         L = L-1; n = n-s-1
30:         yield (N-1) - n
```

Proof. We have $B_i = \begin{cases} 0 & \text{with probability } 1-p_i \\ 1 & \text{with probability } p_i \end{cases}$

If we pick $q \geq \max_{i=1}^{n} p_i$ so that $x > q \implies x > p_i$ for any p_i, and generate for each p_i the standard uniform variates $U_i \sim U(0,1)$ and $V_i \sim U(0,1)$, then we can observe the previous probabilities by using

$$B_i = \begin{cases} 0 & \text{if } (U_i > q) \quad \vee \quad (U_i \leq q \ \wedge \ V_i > p_i/q) \\ 1 & \text{if } (U_i \leq q) \quad \wedge \quad (V_i \leq p_i/q) \end{cases}$$

If $U_i > q$, we do not need to generate the variate V_i and since q is independent of i we can draw a geometric variate $G \sim \text{Geom}(q)$ (that gives the number of failures) to determine how often the case $U_i \geq q$ occurs consecutively before we observe $U_i < q$, which obviates the need for generating individual U_i variates. Each such geometric variate can be obtained using $G \sim \lfloor \log_{1-q} U(0,1) \rfloor$ and since $\mathrm{E}[G] = \frac{1}{q} - 1$ and the expected jump in each iteration is incremented to $\frac{1}{q}$ we expect to draw $\lceil qn \rceil$ geometric and uniform variates in expectation. □

Algorithm 3: Sequential sampling with repl. $F^{-1}(y) = \lfloor yN \rfloor$ for the discrete uniform distribution between 0 and $N-1$.

$\alpha = 1$
while $n > 0$ **do**
  $m \sim \sqrt[N]{U(0,1)}$
  $\alpha = \alpha \cdot m$
  Report item $F^{-1}(1-\alpha)$
  $N = N - 1$

The quantity $H \sim n - \sum_{i=0}^{n-1} B_i$ is calculated where each $B_i \sim \text{Bernoulli}(p_i)$ with $p_i = 1 - \frac{N-n}{N-i}$. We then pick $q \leq 1 - \frac{N-n}{N} = \frac{n}{N}$ (cf. Code 1, line 8) which is at least as large as any remaining p_i, and apply Theorem 3.1 to obtain an average case complexity of $O(n\frac{n}{N})$ for all operations in Step 1.

Step 2: Sequential With-Replacement This step draws H high-items with-replacement in descending order (cf. Code 1, lines 15-23) and uses the mirror trick to report them globally in ascending order (cf. Code 1, line 20). With-replacement duplicates are handled by counting them towards the number of low-items (cf. Code 1, line 22).

Any discrete distribution can be sampled by generating random variates between 0 and 1 and applying the inverse probability transform. We can think of a uniform discrete distribution's cumulative distribution (CDF) F with K possible outcomes as a step-function (piecewise constant function) over the real line whose inverse function is equal to $F^{-1}(y) = \lfloor yK \rfloor$ such that $F^{-1}(y)$ is an index between 0 and $K-1$ that corresponds to each possible outcome. We make use of this in Algorithm 3, along with the fact that standard uniform variates can be drawn in descending/ascending order using the following (well-known) observation:

Lemma 3.2. *Let $U_i \sim U(0,1)$ and $V_i \sim U(0,1)$ be i.i.d. uniform random variables. Then the k-th largest amongst $V_1, \ldots, V_N$ follows the same distribution as $\sqrt[k]{U_k} \cdot \sqrt[k+1]{U_{k+1}} \cdots \sqrt[N]{U_N}$.*

Proof. Let an (a,b)-uniform variable be a continuous uniform variable between $a \in [0,1]$ and $b \in [a,1]$. The probability that a $(0,1)$-uniform variable is below a certain value $x \in [0,1]$ is x. Thus, the probability that n independent $(0,1)$-uniform variables are all below $x \in [0,1]$ is x^N. The cumulative probability function for the maximum of n independent $(0,1)$-uniform variables is therefore $F(x) = x^N$ and the inverse $F^{-1}(y) = y^{1/N}$. Thus, $X \sim U(0,1)^{1/N}$ is distributed like the maximum of n independent $(0,1)$-uniform variables. To obtain the maximum of k independent $(0,b)$-uniform variables from $(0,1)$-uniform

variables one can rescale by b, i.e., $X \sim b \cdot U(0,1)^{1/k}$ will follow the same distribution as the maximum of k independent (0,b)-uniform variables. The second-largest value is then the maximum of $N-1$ independent $(0, V_N)$-uniform variables where V_N is the previously obtained maximum. The claim then follows by repeating this step until the smallest item is reached.

□

Step 3: Sequential Without-Replacement for Small Populations. This step draws L low-items without-replacement in descending order (cf. Code 1, lines 24-30) and uses the mirror trick to report them globally in ascending order (cf. Code 1, line 30). We proceed here along the same lines as Algorithm A (Vitter, 1984), but simplify some formulas to make it easier to validate. Order statistics for without-replacement samples (Nagaraja, 1992) allow us to derive the probability that a gap of size s occurs between adjacent sampled items. The derivation observes that for N remaining items and n remaining samples there exist $\binom{N}{n}$ subsets that are equally probable continuations of our sample. If s is the gap size and we select the item after the gap, there will be afterwards $N-s-1$ remaining items and $n-1$ remaining samples, which means there are $\binom{N-s-1}{n-1}$ equally probable continuations after such a gap. This also implies there are that many subsets that feature such a gap. Thus, the probability of the gap being equal to s is $f(s) := \binom{N-s-1}{n-1}/\binom{N}{n}$ for N remaining items and n remaining samples. In order to draw the size of the gap between selected items, we need to sample from a distribution with cumulative distribution function $F(s) = \sum_{k=0}^{s} f(k) = 1 - \binom{N-s-1}{n}/\binom{N}{n}$. It can then be deduced that the CDF satisfies $F(0) = \frac{n}{N}$ and $F(s+1) = 1 - (1 - \frac{n}{N-s})(1 - F(s))$ for $s > 0$ (cf. Code 1, line 27) in order to facilitate a sequential search for F^{-1} (cf. Code 1, lines 25-28).

3.3 Analysis

Theorem 3.3 (correctness)**.** *The Hidden Shuffle algorithm in Code 1 generates a simple random sample of size n sequentially from a population of size N.*

Proof. The algorithm simulates a Knuth shuffle (cf. Algorithm 1) to obtain a (sequential) sample over the high-items ($\geq n$) and then draws the remaining low-items ($< n$) separately.

In Step 1 it simulates how many positions larger than $n-1$ will be swapped with the first positions. In Step 2 it obtains all high-items that will be swapped into the sample. In Step 3 it obtains a sample over the low-items. □

Theorem 3.4 (average-case variates & time)**.** *The Hidden Shuffle algorithm in Code 1 is expected to generate $n(1+\frac{3n}{N})$ standard uniform variates. Each sampled low-item $x \leq n$ is selected based on a uniform variate that is processed on average in $O(\frac{N}{n})$ operations. Any other variates are processed in $O(1)$.*

Proof. On average $\frac{2n^2}{N}$ variates are generated in step 1, and $\mathrm{E}[H+L] = n + \frac{2n^2}{N}$ variates in step 2 and 3. The inverse transform look-up of Algorithm A in step 3 performs on average $O(\frac{N}{n})$ operations. □

Theorem 3.5 (worst-case variates & time)**.** *The Hidden Shuffle algorithm in Code 1 generates at most $4n$ standard uniform variates and performs $O(n)$ arithmetic operations in total.*

Proof. In the worst-case Step ① generates n pairs of uniform variates, where one is used for the geometric distribution and the other for the residual check. Step ② and Step ③ each generate in the worst-case n uniform variates. Step ① and ② process each variate in $O(1)$ and in step ③ there are $O(1)$ operations for each of the n low-items. □

Theorem 3.6 (worst-case memory)**.** *At any point the Hidden Shuffle algorithm in Code 1 needs to store at most 7 values.*

Proof. During step 1 it uses the seven values `N` and `n`, `H`, `i`, `i_old`, `q` and `p_i`. During step 2 it also uses seven values, i.e., `N` and `n`, `H`, `a`, `S_old`, `S` and `L`. During step 3 it only uses the six values `N` and `n`, `L`, `u`, `F` and `s`. □

4 Experimental Evaluation

Experimental Setup. Due to the availability of relevant baselines in standard libraries of `Python` and `D`, we use Code 1 to implement Hidden Shuffle in `Python`[4] and port it almost line-for-line to `C++`[5] and the less well-known language `D`[6]. Measurements are obtained using the command `/usr/bin/time -v` on Ubuntu 18.04.4 using one of 16 cores of the same machine with an Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz. All our implementations have been written by the same author and were validated by inspecting the minima, maxima and gap distributions of collected samples. The code is available on `https://github.com/shekelyan/sampleiterator`.

[4]Executed by `Python` 3.7 or `PyPy` 7.3.2 (`www.pypy.org`).
[5]Compiled with GCC 7.50 and flags `-O3 -std=c++11`.
[6]Executed with DUB v.1.22.0 (`www.dlang.org`).

implementation	$n = 10^6$	$n = 10^7$	$n = 10^8$
(Python) *Sorted-Duplicate-Rejection*	1.1s (112 MB)	14s (0.9 GB)	243s (38.5 GB)
(Python) *Duplicate-Rejection*	**0.64s** (112 MB)	**6.9s** (0.9 GB)	978s (38.5 GB)
(Python) Hidden Shuffle method	0.83s (12 MB)	8.1s (12 MB)	**87s** (12 MB)
(PyPy) *Sorted-Duplicate-Rejection*	1.3s (167 MB)	19s (1.0 GB)	212s (12.8 GB)
(PyPy) *Duplicate-Rejection*	0.48s (154 MB)	5.3s (0.8 GB)	39s (10.5 GB)
(PyPy) Hidden Shuffle method	**0.26s** (70 MB)	**1.3s** (70 MB)	**12s** (72 MB)
(D) *Vitter's Algorithm D*	0.92s (159 MB)	2.9s (159 MB)	15s (159 MB)
(D) Hidden Shuffle method	**0.82s** (159 MB)	**1.7s** (159 MB)	**11s** (159 MB)
(C++) *Sorted-Floyd's*	0.36s (49 MB)	5.3s (464 MB)	61s (4.5 GB)
(C++) *Floyd's*	0.33s (49 MB)	4.6s (464 MB)	53s (4.5 GB)
(C++) Hidden Shuffle method	**0.07s** (4 MB)	**0.71s** (4 MB)	**7.5s** (4 MB)

Table 2: Comparison of runtime and memory usage for population size $N = 10^9$ and varied sample size n.

Baselines. *Algorithm D* by Vitter (1984) is the state-of-the-art of sequential sampling techniques. We use the standard library implementation of `D` to compare with it. *Duplicate-Rejection* is a commonly used technique across various standard libraries[7]. Items are drawn with-replacement and duplicates are rejected with the help of a hash map to obtain a without-replacement sample. The rejection rate grows rapidly when the sample size gets close to the population size, requiring reservoir sampling as a fallback. We use `Python`'s standard library function `random.sample` to compare with this method. *Floyd's* sampling algorithm is an elegant improvement by Bentley and Floyd (1987) without rejections. The variants *Sorted-Duplicate-Rejection* and *Sorted-Floyd's* sort the sampled positions first to report samples sequentially.

Runtime and Memory Usage. Table 2 shows extensive comparisons with the baselines. As expected Hidden Shuffle and *Vitter's Algorithm D* require vastly less memory than other approaches. Hidden Shuffle is almost an order of magnitude faster than *Floyd's*, slightly faster than *Vitter's Algorithm D* and comparably fast to `Python`'s standard library implementation of *Duplicate-Rejection*. Sequential baselines based on sorting are a lot slower. Even a `Python` implementation of Hidden Shuffle is nearly as fast as *Sorted-Floyd's* in `C++`. Due to the unusual slowdown of *Duplicate-Rejection* for $n = 10^8$, we also tried running the `Python` code with the alternative interpreter `PyPy`, which achieves speed-ups through just-in-time compilation. Using `PyPy` the aforementioned slowdown cannot be observed, the performance almost matches precompiled languages and Hidden Shuffle emerges as the fastest approach.

[7]The *Duplicate-Rejection* technique is for instance employed in `random.sample` in Python (`www.python.org`) or `sample.int` in R (`www.r-project.org`).

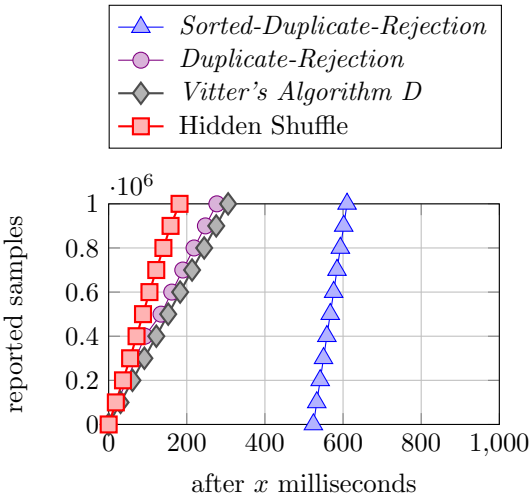

Figure 2: Timeline for how many sampled elements are reported after x milliseconds for a population size $N = 10^9$ and sample size $n = 10^6$.

Latency results and number of variates. Next, two sampling instances are examined in more detail, where to compare the throughput and variate usage of the different sampling approaches (implemented in `D` to facilitate a comparison with *Vitter's Algorithm D*). Unlike for Table 2 the measurements are here conducted from within `D` to obtain a temporal breakdown. Figure 2 shows the timeline for a million and Figure 3 (cf. next page) for a billion sampled items, in each case for a thousand times larger population. In both instances, Hidden Shuffle reports the samples the fastest and at a steady pace, closely followed by *Vitter's Algorithm D* and *Duplicate-Rejection*. *Sorted-Duplicate-Rejection* is visibly delayed as it has to first collect and sort the full sample before it can report any samples. All four approaches generated in these experiments at most $1.02n$ variates for a sample size of n.

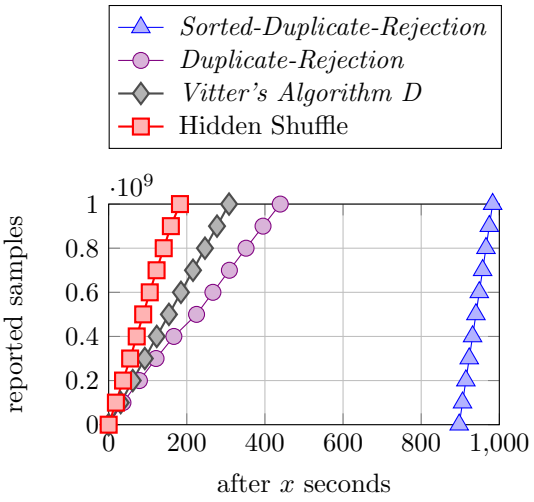

Figure 3: Timeline for how many sampled elements are reported after x seconds for a population size $N = 10^{12}$ and sample size $n = 10^9$.

5 Concluding Remarks

The proposed Hidden Shuffle method is shown to posses many attractive features: it is very lightweight, can be implemented with few lines of code, has strong theoretical guarantees and shows uniformly superior performance in practice. While the focus of this work is simple random sampling, the new techniques could also pave the way for solutions of more challenging sampling problems. For instance, random partitions for fixed block sizes could be obtained by maintaining hierarchies of samples, since the memory and computation footprint is so small. Furthermore, the basic idea of how to reduce without-replacement to with-replacement sampling could make it easier to deal with sampling problems in a distributed setting, where data is spread over multiple machines.

Acknowledgements. This work is supported by European Research Council grant ERC-2014-CoG 647557.

References

Ahrens, J. H. and Dieter, U. (1985). Sequential random sampling. *ACM Trans. Math. Softw.*, 11(2):157–169.

Bebbington, A. (1975). A simple method of drawing a sample without replacement. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 24(1):136–136.

Bentley, J. and Floyd, B. (1987). Programming pearls: a sample of brilliance. *Communications of the ACM*, 30(9):754–757.

Bui, D. N. (2015). Cachediff: Fast random sampling. *CoRR*, abs/1512.00501.

Cormode, G., Muthukrishnan, S., Yi, K., and Zhang, Q. (2010). Optimal sampling from distributed streams. In *Symposium on Principles of Database Systems (PODS)*, pages 77–86. ACM.

Devroye, L. (1986). *Non-Uniform Random Variate Generation*. Springer.

Efraimidis, P. S. and Spirakis, P. G. (2006). Weighted random sampling with a reservoir. *Inf. Process. Lett.*, 97(5):181–185.

Fan, C., Muller, M. E., and Rezucha, I. (1962). Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 57(298):387–402.

Jones, T. G. (1962). A note on sampling a tape-file. *Communications of the ACM*, 5(6):343.

Knuth, D. E. (1969). *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley.

Li, K.-H. (1994). Reservoir-sampling algorithms of time complexity $O(n(1 + \log(N/n)))$. *ACM Transactions on Mathematical Software (TOMS)*, 20(4):481–493.

Nagaraja, H. (1992). Order statistics from discrete distributions. *Statistics*, 23(3):189–216.

Sanders, P., Lamm, S., Hübschle-Schneider, L., Schrade, E., and Dachsbacher, C. (2018). Efficient parallel random sampling - vectorized, cache-efficient, and online. *ACM Trans. Math. Softw.*, 44(3):29:1–29:14.

Stadlober, E. (1990). The ratio of uniforms approach for generating discrete random variates. *Journal of computational and applied mathematics*, 31(1):181–189.

Sunter, A. (1977). List sequential sampling with equal or unequal probabilities without replacement. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 26(3):261–268.

Tirthapura, S. and Woodruff, D. P. (2011). Optimal random sampling from distributed streams revisited. In *Distributed Computing DISC*, pages 283–297. Springer.

Vitter, J. S. (1984). Faster methods for random sampling. *Commun. ACM*, 27(7):703–718.

Vitter, J. S. (1985). Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57.

Vitter, J. S. (1987). An efficient algorithm for sequential random sampling. *ACM Trans. Math. Softw.*, 13(1):58–67.