Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SHORT COMMUNICATION

Preprint (submitted version). Cite the version of record as:
Cicirello, V.A. Algorithms for generating small random samples. *Software: Practice and Experience*. 2024; 1–9. https://doi.org/10.1002/spe.3379

Algorithms for Generating Small Random Samples

Vincent A Cicirello

[1]Computer Science, Stockton University, New Jersey, USA

Correspondence
Vincent A Cicirello, Computer Science, School of Business, Stockton University, 101 Vera King Farris Dr, Galloway, NJ 08205
Email: vincent.cicirello@stockton.edu

Present address
Vincent A Cicirello, Computer Science, School of Business, Stockton University, 101 Vera King Farris Dr, Galloway, NJ 08205
Email: vincent.cicirello@stockton.edu

Abstract

We present algorithms for generating small random samples without replacement. We consider two cases. We present an algorithm for sampling a pair of distinct integers, and an algorithm for sampling a triple of distinct integers. The worst-case runtime of both algorithms is constant, while the worst-case runtimes of common algorithms for the general case of sampling k elements from a set of n increase with n. Java implementations of both algorithms are included in the open source library $\rho\mu$.

KEYWORDS
algorithm, random sampling, Java, open source, small samples

1 | INTRODUCTION

Efficiently generating random samples of k elements from a set of n elements, without replacement, is important to a variety of applications. There are algorithms for handling this general case, such as the reservoir sampling family of algorithms[1,2], pool sampling[3], insertion sampling[4], among others[5,6]. The runtime of pool sampling and some forms of reservoir sampling is $O(n)$. Other forms of reservoir sampling[2] improve the runtime to $O(k(1 + \ln(n/k)))$, but this still increases with n. Insertion sampling was designed with small samples in mind, with a runtime of $O(k^2)$, requiring k random numbers to generate a sample of size k. It is an efficient choice when you know k will be small, although for large k it is much slower than alternatives. Others such as Ting make other improvements to the classical random sampling algorithms[7]. There are also several parallel algorithms for random sampling without replacement[8], weighted random sampling[9], and for random sampling (both weighted and unweighted) from data streams[10,11,12].

Although insertion sampling handles small k nicely, we can do much better with algorithms designed for the specific value of k. For example, the special case of $k = 1$ has been widely studied, such as Lemire's algorithm for random integers in an interval that is significantly faster than the alternatives provided within common programming languages by avoiding nearly all division operations[13], or Goualard's approach to sampling random floating-point numbers from an interval[14]. It is common for applications to require repeatedly generating random integers from an interval, such as algorithms for shuffling arrays, as well as sampling algorithms for the general case. Brackett-Rozinsky and Lemire propose an algorithm for sampling multiple independent bounded random integers from a single random word, and demonstrate how the technique can speed up algorithms for shuffling[15].

In this article, we present algorithms for the special cases of $k = 2$ and $k = 3$. The runtime of both algorithms is $O(1)$, with the algorithm for generating a random pair of distinct integers requiring two random numbers, and the algorithm for generating a random triple of distinct integers requiring three random numbers. Our original motivation for efficient algorithms for randomly sampling pairs and triples of distinct integers is in the implementation of various mutation and crossover operators for evolutionary algorithms for the space of permutations[16], where such evolutionary operators require generating random combinations of indexes. In such an application, one requires random samples from the integer interval $[0, n)$. Thus, the

Abbreviations: JMH, Java microbenchmark harness; JVM, Java virtual machine; PRNG, pseudorandom number generator.

Algorithm 1 RandomPair(n)

```
1: i ← Rand(n)
2: j ← Rand(n – 1)
3: if j = i then j ← n – 1
4: return (i, j)
```

Algorithm 2 RandomTriple(n)

```
1: i ← Rand(n)
2: j ← Rand(n – 1)
3: k ← Rand(n – 2)
4: if k = j then k ← n – 2
5: if j = i then j ← n – 1
6: if k = i then k ← n – 1
7: return (i, j, k)
```

algorithms that we present are specified as sampling from that interval. However, without loss of generality, both algorithms are applicable for the broader case of sampling pairs and triples of elements from a set of n elements, provided you define a mapping of the n elements to the integers in $[0, n)$. We utilize Lemire's algorithm for bounded random integers [13] within our implementation of the algorithms of this paper, as well as in our implementations of the general sampling algorithms used in the experiments. The Brackett-Rozinsky and Lemire approach to multiple bounded random integers [15] can potentially enable further optimizations as well, but we have not explored doing so.

The two algorithms for sampling pairs and triples of integers are presented in Section 2. We also discuss in that section how one can derive a special purpose sampling algorithm for any fixed k in the form of a sampling network, although for larger k this is likely impractical, and demonstrate with an algorithm for the case of $k = 4$. We provide Java implementations in the open source library $\rho\mu$, which provides a variety of randomization-related utilities [17]. We empirically compare the runtime performance of the algorithms to existing random sampling algorithms using the $\rho\mu$ library, describing our experimental methodology in Section 3 and the results in Section 4. We wrap up in Section 5.

2 | ALGORITHMS

We now present the algorithms. In our pseudocode, the function Rand(n) returns a random integer uniformly distributed in the half-open interval $[0, n)$.

Algorithm 1, RandomPair(n), generates a random pair (i, j) of distinct integers from the interval $[0, n)$, uniformly distributed over the space of all $2\binom{n}{2}$ distinct pairs. Line 1 generates the first of the two integers, i, uniformly from the interval $[0, n)$. Then, a random integer j uniformly distributed over the interval $[0, n-1)$ is generated (line 2); and if j is the same as i, we reset j to $n-1$ (line 3), since $n-1$ was excluded via the half-open interval $[0, n-1)$.

Algorithm 2, RandomTriple(n), generates a random triple (i, j, k) of distinct integers from the interval $[0, n)$, uniformly distributed over the space of all $6\binom{n}{3}$ distinct triples. It initially generates i, j, and k uniformly at random from the half-open intervals $[0, n)$, $[0, n-1)$, and $[0, n-2)$, respectively. The remainder of the algorithm maps duplicates to the values excluded by the half-open intervals. Line 4 ensures that j and k are different. Lines 5–6 in turn adjusts if necessary so that neither are the same as i.

The worst-case runtime of both algorithms is $O(1)$. Algorithm 1 and Algorithm 2 generate 2 and 3 random integers, respectively; and both have constant numbers of comparisons and assignments to adjust for duplicates.

The bounds of the random numbers utilized by Algorithm 1 and Algorithm 2 are similar to those of the first two and three steps of a Fisher-Yates shuffle [18] as well as the first two to three iterations of insertion sampling (see later in Section 3.1). Additionally, each if-statement that resolves duplicates is reminiscent of the "compare-exchange" operation of sorting networks [19]. But, instead of a "compare-exchange," it is a "compare-change." This suggests the potential to derive a *sampling network* from the approach. Such a sampling network may have similar benefits as sorting networks, such as lending themselves well to hardware implementation and parallelism. For example, the execution order of lines 5 and 6 of Algorithm 2 does not matter. It is

Algorithm 3 RandomFourTuple(n)

```
1:  h ← Rand(n)
2:  i ← Rand(n – 1)
3:  j ← Rand(n – 2)
4:  k ← Rand(n – 3)
5:  if k = j then k ← n – 3
6:  if j = i then j ← n – 2
7:  if k = i then k ← n – 2
8:  if i = h then i ← n – 1
9:  if j = h then j ← n – 1
10: if k = h then k ← n – 1
11: return (h, i, j, k)
```

Algorithm 4 ReservoirSamplingR(n, k)

```
1: sample ← a new array of length k
2: for i = 0 to k – 1 do sample[i] ← i
3: for i = k to n – 1 do
4:     j ← Rand(i + 1)
5:     if j < k then sample[j] ← i
6: end for
7: return sample
```

straightforward to derive a corresponding sampling network for any fixed k, although the benefit in software diminishes since the obvious approach leads to a number of compare-changes that increases quadratically in k. For example, Algorithm 3 shows how to extend the approach to $k = 4$. A hypothetical circuit implementation could align the compare-changes in layers of independent operations, such as lines 6–7 in parallel and lines 8–10 in parallel.

3 | EXPERIMENTAL METHODOLOGY

3.1 | General Sampling Algorithms

We experimentally validate the approach relative to existing algorithms for the general case of sampling k distinct elements from a set of n to explore the benefit of special purpose algorithms when k is known and small.

First, we consider two forms of reservoir sampling [1,2], which refers to a family of sampling algorithms. Reservoir algorithms were originally introduced decades ago for the problem of randomly sampling k records from an unknown number of records n via a single pass over the records. But they are also easily adapted to the simpler problem of sampling integers with known n. The first reservoir sampling algorithm that we consider is the simplest, and referred to by Vitter as algorithm R [1]. We provide pseudocode in Algorithm 4. It begins by initializing the sample with the first k elements (lines 1–2). It then iterates over the remaining $n - k$ elements in a very similar way to an array shuffling procedure, choosing a random index j for the next element (lines 3–4). If that index is in the bounds of the sample, that element replaces the one at that index in the sample (line 5). The runtime of this reservoir sampling algorithm is $O(n)$, and it requires $O(n - k)$ random numbers, which is desirable when k is large, but there are better algorithms for small k. In the same article, Vitter described a series of improvements leading to his algorithms X, Y, and Z, the last of which having an expected runtime of $O(k(1 + \ln(n/k)))$.

The other reservoir sampling algorithm that we consider is Li's algorithm L [2], which is simpler and an improvement over Vitter's algorithm Z [1]. We provide pseudocode for the L form of reservoir sampling in Algorithm 5. In the pseudocode, $U()$ returns a random floating-point value in $[0.0, 1.0)$. It begins like the previous reservoir algorithm by initializing the sample with the first k elements (lines 1–2). Instead of explicitly iterating over all $n - k$ remaining elements, it skips groups of elements. See Li for derivation of the skip lengths and the details of the runtime analysis [2]. The expected runtime still grows with n, but much

Algorithm 5 ReservoirSamplingL(n, k)

```
1: sample ← a new array of length k
2: for i = 0 to k – 1 do sample[i] ← i
3: w ← exp(ln(U())/k)
4: i ← k + ⌊ln(U())/ln(1–w)⌋
5: while i < n do
6:     sample[Rand(k)] ← i
7:     w ← w · exp(ln(U())/k)
8:     i ← i + 1 + ⌊ln(U())/ln(1–w)⌋
9: end while
10: return sample
```

Algorithm 6 PoolSampling(n, k)

```
1: sample ← a new array of length k
2: pool ← a new array of length n
3: for i = 0 to n – 1 do pool[i] ← i
4: remaining ← n
5: for i = 0 to k – 1 do
6:     j ← Rand(remaining)
7:     sample[i] ← pool[j]
8:     remaining ← remaining – 1
9:     pool[j] ← pool[remaining]
10: end for
11: return sample
```

better than linearly, $O(k(1 + \ln(n/k)))$. As k approaches n, this converges to $O(k)$. Thus, this is an especially good choice for larger k. There are k steps to initialize the sample, and the expected number of iterations of the while loop is $O(k \ln(n/k))$. It should be noted that the iterations are more costly with each iteration generating two uniform random floating-point values and a random bounded integer, along with an exponentiation and three logs.

Next consider pool sampling[3], which has additional names in the literature, including random draw sampling[20]. Its authors originally referred to it generically as "SELECT"[3], but there are others with that same name. Algorithm 6 shows it in pseudocode form. It maintains a pool of unused elements (initialized in lines 2–4 and updated in lines 8–9), and draws elements randomly from the pool (lines 6–7) until the sample is complete. Pool sampling requires k random numbers to generate a sample of size k, which is good when k is small, however its runtime grows linearly in n; and it also requires $O(n)$ extra storage. Ernvall and Nevalainen[5] use a hash table to reduce the extra storage to $O(k)$ and the runtime to $O(k^2)$.

Like pool sampling, the insertion sampling[4] algorithm requires only k random numbers. But unlike pool sampling, its storage requirement is constant. Its runtime is $O(k^2)$. It is not widely studied as it was introduced within the context of a specific application, one where k is unknown ahead of time but likely small. Algorithm 7 provides the details. It maintains the currently sampled elements in sorted order (lines 5, 7–10). It iteratively generates a sequence of random integers $v \in [\text{Rand}(n), \text{Rand}(n-1), \ldots, \text{Rand}(n-k+1)]$ (line 3). Each v is treated as an index into an ordered (implicit) list of the unused elements. Line 6 skips over used elements as v is inserted into the sample in its sorted position (line 10).

The proposed algorithms in Section 2 (Algorithms 1, 2, and 3) produce samples that are not only uniformly random over combinations, but also uniformly random over permutations. This is also true of pool sampling (Algorithm 6). However, it is not true of the other reference algorithms. Insertion sampling (Algorithm 7) generates an ordered sample. For uniform ordering, shuffle the sample before the return on line 12. Line 2 of both versions of reservoir sampling (Algorithms 4 and 5) constrains the possible positions of the first k elements. Shuffling the initial sample of line 2 achieves uniform random ordering. In all three cases, the adjustment introduces an $O(k)$ step with $O(k)$ additional random numbers. In the experiments, we use these reference algorithms as specified in the pseudocode and do not introduce such shuffling steps.

Algorithm 7 InsertionSampling(n, k)

```
1: sample ← a new array of length k
2: for i = 0 to k – 1 do
3:     v ← Rand(n – i)
4:     j ← k – i
5:     while j < k and v ≥ sample[j] do
6:         v ← v + 1
7:         sample[j – 1] ← sample[j]
8:         j ← j + 1
9:     end while
10:    sample[j – 1] ← v
11: end for
12: return sample
```

TABLE 1 Important URLs for the $\rho\mu$ open source library and experiments

$\rho\mu$ library	
Source	https://github.com/cicirello/rho-mu
Website	https://rho-mu.cicirello.org/
Maven	https://central.sonatype.com/artifact/org.cicirello/rho-mu/
Experiments	
Source/data	https://github.com/cicirello/small-sample-experiments

TABLE 2 Methods of $\rho\mu$'s EnhancedRandomGenerator class used in the experiments

Algorithm	Method of EnhancedRandomGenerator class
RandomPair (Algorithm 1)	nextIntPair
RandomTriple (Algorithm 2)	nextIntTriple
insertion sampling [4]	sampleInsertion
pool sampling [3]	samplePool
reservoir (R) sampling [1]	sampleReservoir
reservoir (L) sampling [2]	ReservoirL class in experiment repository

3.2 | Methodology

We implemented the new algorithms from Section 2, and three of the other sampling algorithms from Section 3.1 in the open source library $\rho\mu$. The experiments use $\rho\mu$ 4.2.0. The code for reservoir (L) sampling is available in the GitHub repository that contains the code to reproduce the experiments, and the raw data from my runs. See Table 1 for URLs to the library source code, project website, and in the Maven Central repository, as well as for the source code and raw data from the experiments.

Table 2 lists the methods of $\rho\mu$'s EnhancedRandomGenerator class that are used in the experiments. The EnhancedRandomGenerator class wraps any class that implements Java 17's RandomGenerator interface adding a variety of functionality to it, or in some cases replacing implementations with more efficient algorithms. There are multiple nextIntPair and nextIntTriple methods, which differ in terms of return type (e.g., returning an array of type int versus returning a library specific record IndexPair and IndexTriple).

We use OpenJDK 64-Bit Server VM version 17.0.2 in the experiments on a Windows 10 PC with an AMD A10-5700, 3.4 GHz processor and 8GB memory. For the pseudorandom number generator (PRNG), we use Java's SplittableRandom class, which implements the SplitMix [21] algorithm, which is a faster optimized version of the DotMix [22] algorithm, and which passes the DieHarder [23] tests.

The $\rho\mu$ library's EnhancedRandomGenerator class replaces Java's method for generating random integers subject to a bound with an implementation of a much faster algorithm [13]. Our prior experiments [17] show that Lemire's approach uses less than half the CPU time as Java's RandomGenerator.nextInt(bound) method. So although we are using Java's SplittableRandom class as

TABLE 3 Sampling pairs of distinct integers: average time (ns) with 99.9% confidence intervals

n	RandomPair (Algorithm 1)	insertion[4]	pool[3]	reservoir (R)[1]	reservoir (L)[2]
$n = 16$	16.0 ± 0.5	$24.1 \pm <0.1$	54.9 ± 0.3	178.2 ± 1.0	465.8 ± 14.5
$n = 64$	15.6 ± 0.1	24.1 ± 0.1	89.3 ± 1.9	633.4 ± 9.7	721.3 ± 19.5
$n = 256$	15.7 ± 0.6	$24.3 \pm <0.1$	298.8 ± 0.8	2062.5 ± 9.7	981.3 ± 8.4
$n = 1024$	$15.7 \pm <0.1$	23.6 ± 0.1	1209.9 ± 16.2	7918.2 ± 12.9	1242.7 ± 1.4

the PRNG, by wrapping it in an instance of $\rho\mu$'s EnhancedRandomGenerator, we are significantly speeding the runtime of all of the algorithms in our experiments.

We use the Java Microbenchmark Harness (JMH)[24] to implement our experiments. JMH is developed by the same team that develops the OpenJDK, and is the preferred harness for benchmarking Java code. Since the Java Virtual Machine (JVM) is adaptive, utilizing just-in-time compilation of hot-spots identified at runtime to boost performance, it is important to warm up the JVM to achieve a steady-state before benchmarking. The JMH handles this warmup phase, and other Java benchmarking related challenges. For each experiment condition (e.g., algorithm and value of n) we use five 10-second warmup iterations to ensure that the Java JVM is properly warmed up, and we likewise use five 10-second iterations for measurement. JMH executes the benchmark method as many times as each 10-second iteration allows, which leads to hundreds of millions of method invocations for each experiment condition. We use JMH's built-in blackhole to consume the generated samples to prevent dead-code elimination. We measure and report average time per operation in nanoseconds, along with 99.9% confidence intervals as calculated by JMH. The times reported by JMH are not CPU times. Although not indicated in the JMH documentation, examination of the JMH source code[24] reveals that it calculates elapsed time using the JVM's high-resolution time source via the System.nanoTime() method, which returns nanoseconds since an arbitrary origin. We round the averages to the nearest tenth of a nanosecond since the JVM's time source is of nanosecond precision (JMH's reported averages are to thousandths of a nanosecond, a false level of precision). A consequence of elapsed time is that benchmarks can be affected by the state of the system (e.g., competition for system resources). To minimize such impact, we shut down all applications under our control, and disabled various non-critical background processes (e.g., auto-updaters, etc) to limit background processes as much as feasible.

4 | RESULTS

4.1 | Sampling Pairs of Distinct Integers

This first set of results in Table 3 explores the difference in runtime performance of the various approaches to sampling random pairs of distinct integers from the half open interval $[0, n)$. It is clear from the results that the RandomPair (Algorithm 1) of this paper dominates all of the others. Its average runtime is a low constant, and does not vary as n increases, at least not to any significant degree. If it appears that average time is possibly decreasing as n increases for this algorithm, it may be, although it is clearly negligible. This is because as n increases, the probability of the conditional assignment on line 3 of Algorithm 1 decreases. The insertion sampling[4] average runtime is likewise approximately constant, though a higher constant. Note that insertion sampling's theoretical runtime is $O(k^2)$, which is essentially constant for any fixed k. Pool sampling[3] and reservoir (R) sampling[1] both have an average runtime that increases linearly in n. Reservoir (R) sampling's average runtime is much higher than pool sampling, however, because reservoir (R) sampling also generates $(n - k)$ random integers to sample k integers from a set of n integers, while pool sampling only needs k random integers to accomplish this. Thus, reservoir (R) is an especially poor choice for low k relative to n. Reservoir (L) sampling[2] is the slowest for low n, but it overtakes reservoir (R) as n increases, and should surpass the performance of pool sampling if we further increase n. Its runtime increases with n, but at a slower rate $O(k(1 + \ln(n/k)))$ than reservoir (R) and pool sampling.

The results discussed above used the version of the EnhancedRandomGenerator.nextIntPair method that allocates and returns an array for the pair of integers, which is also what the various sampling algorithm implementations do. The $\rho\mu$ library also provides a version of the EnhancedRandomGenerator.nextIntPair that returns a Java record. A Java record is immutable and potentially enables the JVM to optimize in ways that it otherwise cannot. To explore this, we ran an additional experiment comparing three cases: (a) allocating and returning a new array for the integer pair, (b) using a preallocated array passed as a parameter, and (c) allocating and returning a record instead of an array. In this experiment, rather than using JMH's blackhole to consume the generated pairs directly (as we did in the first set of results), we simulate an operation on the pair of integers to give

TABLE 4 Sampling pairs of distinct integers: average time (ns) with 99.9% confidence intervals

n	returning new array	preallocated array	Java record
$n = 16$	16.1 ± 0.1	13.7 ± 0.2	$12.4 \pm <0.1$
$n = 64$	$15.5 \pm <0.1$	13.3 ± 0.1	12.0 ± 0.3
$n = 256$	15.6 ± 0.5	13.2 ± 0.2	11.8 ± 0.1
$n = 1024$	15.4 ± 0.1	13.2 ± 0.4	11.9 ± 0.5

TABLE 5 Sampling triples of distinct integers: average time (ns) with 99.9% confidence intervals

n	RandomTriple (Algorithm 2)	insertion[4]	pool[3]	reservoir (R)[1]	reservoir (L)[2]
$n = 16$	$23.6 \pm <0.1$	$41.8 \pm <0.1$	59.4 ± 0.2	179.3 ± 0.3	583.4 ± 0.8
$n = 64$	22.4 ± 0.3	41.1 ± 0.2	91.3 ± 0.3	652.0 ± 0.7	1026.6 ± 3.4
$n = 256$	$22.4 \pm <0.1$	45.6 ± 0.1	286.0 ± 1.1	2113.4 ± 23.6	1459.3 ± 103.8
$n = 1024$	21.8 ± 0.1	42.3 ± 0.3	1177.5 ± 3.7	8564.1 ± 64.3	1867.6 ± 292.3

TABLE 6 Sampling triples of distinct integers: average time (ns) with 99.9% confidence intervals

n	returning new array	preallocated array	Java record
$n = 16$	23.8 ± 0.1	$21.7 \pm <0.1$	$19.2 \pm <0.1$
$n = 64$	$23.2 \pm <0.1$	20.5 ± 0.1	$18.0 \pm <0.1$
$n = 256$	$23.4 \pm <0.1$	$21.0 \pm <0.1$	18.7 ± 0.2
$n = 1024$	22.8 ± 0.1	20.0 ± 0.5	18.3 ± 0.1

the JVM the opportunity to exploit the immutability property of Java records. Specifically, we do a simple sum of the integers in the pairs, and return that to JMH's blackhole.

Table 4 shows the results of this second experiment. The fastest of the three options uses a Java record type for the pair of integers. It was even faster than using a preallocated array. The difference in time is just a little over a nanosecond per sample on average, so use whichever leads to cleaner code for your use-case.

4.2 | Sampling Triples of Distinct Integers

We now consider the results in Table 5 for sampling random triples of distinct integers. As in the case of sampling pairs of distinct integers, we find that the algorithm of this paper, RandomTriple (Algorithm 2) requires a constant time on average. The insertion sampling algorithm likewise has an average time that is constant, although a higher constant than Algorithm 2. Both pool sampling and reservoir (R) sampling require time that increases linearly in n, with reservoir sampling's times significantly higher than pool sampling due to generating significantly more random numbers during the sampling process for low k. Reservoir (L) sampling is also significantly slower than Algorithm 2.

In Table 6 we further explore the effects of allocating a new array for each sampled triple, versus using a preallocated array, versus returning a Java record. The results in the triple of integers case follow the same pattern as that of the pairs of integers case. For triples of distinct integers, returning a Java record is about 10% faster than using a preallocated array for the result, which in turn is about 10% faster than returning a new array.

5 | CONCLUSIONS

In this article, we presented constant runtime algorithms for sampling pairs and triples of distinct integers from the interval $[0, n)$. We conducted benchmarking experiments with our Java implementations using JMH that demonstrate that these sampling algorithms are substantially faster than using existing general purpose algorithms for sampling k integers from $[0, n)$. For example, reservoir (R) sampling[1] and pool sampling[3] both require linear time, with reservoir (R) sampling being especially computationally expensive for low k since it also requires $n - k$ random integers for sample size k. Reservoir (L) sampling[2] is superior to reservoir (R) for larger n, but it is still very slow relative to the constant time algorithms for the special cases of $k = 2$

and $k = 3$. Although insertion sampling[4] has a constant runtime for any fixed k (e.g., runtime increases with k but not with n), the special purpose algorithms for the two specific cases of $k = 2$ and $k = 3$ presented in this paper are significantly faster.

The structure of Algorithms 1 and 2 for sampling pairs and triples of distinct integers suggest a pattern for forming a special purpose algorithm for any fixed k, such as the example provided for $k = 4$ in Algorithm 3. Although it is unlikely practical or efficient for larger k in software, it is possible that a circuit implementation of such a sampling network may have similar benefits as sorting networks.

Our implementations of the algorithms for small random samples and three of the general purpose sampling algorithms are available in the open source Java library $\rho\mu$. The experiments of this paper, including the fourth general purpose sampling algorithm, are also available on GitHub to assist with reproducibility.

CONFLICT OF INTEREST

The author declares no potential conflict of interests.

REFERENCES

1. Vitter JS. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software.* 1985;11(1):37–57. doi: 10.1145/3147.3165
2. Li KH. Reservoir-sampling algorithms of time complexity O(n(1 + log(N/n))). *ACM Transactions on Mathematical Software.* 1994;20(4):481–493. doi: 10.1145/198429.198435
3. Goodman SE, Hedetniemi ST. *Introduction to the Design and Analysis of Algorithms*ch. 6.3 Probabilistic Algorithms:298–316; New York, NY, USA: McGraw-Hill . 1977.
4. Cicirello VA. Cycle Mutation: Evolving Permutations via Cycle Induction. *Applied Sciences.* 2022;12(11):5506. doi: 10.3390/app12115506
5. Ernvall J, Nevalainen O. An Algorithm for Unbiased Random Sampling. *The Computer Journal.* 1982;25(1):45–47. doi: 10.1093/comjnl/25.1.45
6. Knuth DE. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley. 3rd ed., 1998.
7. Ting D. Simple, Optimal Algorithms for Random Sampling Without Replacement. arxiv preprint arXiv:2104.05091; 2021
8. Sanders P, Lamm S, Hübschle-Schneider L, Schrade E, Dachsbacher C. Efficient Parallel Random Sampling—Vectorized, Cache-Efficient, and Online. *ACM Transactions on Mathematical Software.* 2018;44(3):29. doi: 10.1145/3157734
9. Hübschle-Schneider L, Sanders P. Parallel Weighted Random Sampling. *ACM Transactions on Mathematical Software.* 2022;48(3):29. doi: 10.1145/3549934
10. Tangwongsan K, Tirthapura S. Parallel Streaming Random Sampling. In: Euro-Par 2019: Parallel Processing. Euro-Par. Springer 2019; Cham:451–465.
11. Karras C, Karras A, Drakopoulos G, Tsakalidis K, Mylonas P, Sioutas S. Weighted Reservoir Sampling On Evolving Streams: A Sampling Algorithmic Framework For Stream Event Identification. In: Proceedings of the 12th Hellenic Conference on Artificial Intelligence. SETN. 2022
12. Hübschle-Schneider L, Sanders P. Communication-Efficient Weighted Reservoir Sampling from Fully Distributed Data Streams. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures. ACM. 2020:543-545
13. Lemire D. Fast Random Integer Generation in an Interval. *ACM Transactions on Modeling and Computer Simulation.* 2019;29(1):3. doi: 10.1145/3230636
14. Goualard F. Drawing Random Floating-point Numbers from an Interval. *ACM Transactions on Modeling and Computer Simulation.* 2022;32(3):16. doi: 10.1145/3503512
15. Brackett-Rozinsky N, Lemire D. Batched Ranged Random Integer Generation. arxiv preprint arXiv:2408.06213; 2024.
16. Cicirello VA. A Survey and Analysis of Evolutionary Operators for Permutations. In: Proceedings of the 15th International Joint Conference on Computational Intelligence. ECTA. 2023:288–299
17. Cicirello VA. $\rho\mu$: A Java library of randomization enhancements and other math utilities. *Journal of Open Source Software.* 2022;7(76):4663. doi: 10.21105/joss.04663
18. Durstenfeld R. Algorithm 235: Random permutation. *Communications of the ACM.* 1964;7(7):420–421. doi: 10.1145/364520.364540
19. Batcher KE. Sorting networks and their applications. In: Proceedings of the Spring Joint Computer Conference. ACM. 1968:307–314
20. Bellhouse D, Kulperger R. Computer generated simple random samples. *Communications in Statistics - Simulation and Computation.* 1991;20(2-3):539–550. doi: 10.1080/03610919108812971
21. Steele GL, Lea D, Flood CH. Fast Splittable Pseudorandom Number Generators. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. ACM. ACM 2014:453–472
22. Leiserson CE, Schardl TB, Sukha J. Deterministic Parallel Random-number Generation for Dynamic-multithreading Platforms. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM. ACM 2012:193–204
23. Brown RG, Eddelbuettel D, Bauer D. *Dieharder: A random number test suite*. Duke University; Durham, NC: 2013. https://www.phy.duke.edu/~rgb/General/dieharder.php.
24. OpenJDK.org . Java Microbenchmark Harness (JMH) Version 1.37. https://github.com/openjdk/jmh; 2023.