Algorithms for Generating Small Random Samples

Vincent A. Cicirello

Computer Science
Stockton University
Galloway, NJ 08205 USA
https://www.cicirello.org/

Technical Report ALG-24-008

May 2024

Algorithms for Generating Small Random Samples

Vincent A. Cicirello

Computer Science
Stockton University
Galloway, NJ 08205 USA
https://www.cicirello.org/

Technical Report ALG-24-008

Abstract

This report presents algorithms for generating small random samples without replacement. It considers two cases. It presents an algorithm for sampling a pair of distinct integers, and an algorithm for sampling a triple of distinct integers. The worstcase runtime of both algorithms is constant, while the worstcase runtime of common algorithms for the general case of sampling k elements from a set of n are linear in n. Java implementations of both algorithms are included in the open source library $\rho\mu$.

Keywords: algorithm; random sampling; Java; open source

ACM Classes: F.2.1; I.1.2; D.2.13; G.3; G.4

MSC Classes: 68Q25; 68Q87; 68W20; 68W40; 60-04

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 reservoir sampling [11], pool sampling [5], and insertion sampling [2]. Knuth [6] discusses a variety of algorithms as well. The runtime of both reservoir sampling and pool sampling is $O(n)$. While generating the sample, reservoir sampling requires $(n-k)$ random integers, many more than the sample size k when k is small. It is a great choice when k is large, however. 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 pool sampling also requires $O(n)$ extra memory. 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 the reservoir and pool sampling algorithms.

Although the insertion sampling algorithm handles small k nicely, we can do much better with algorithms designed specially for the specific value of k. In this report, 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 2 random numbers, and the algorithm for generating a random triple of distinct integers requiring 3 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

https://orcid.org/0000-0003-1072-8559

Algorithm 1 $\mathrm{RandomPair}(n)$

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

crossover operators for evolutionary algorithms for the space of permutations [4], where such evolutionary operators require generating random combinations of indexes. In such an application, one needs to generate random samples from the integer interval $[0, n)$. Thus, the 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 $[0, n)$.

The two algorithms for sampling pairs and triples of integers are presented in Section 2. We provide Java implementations in the open source library $\rho\mu$, which provides a variety of randomization-related utilities [3]. We empirically compare the runtime performance of the algorithms compared 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 $\mathrm{Rand}(n)$ returns a random integer uniformly distributed in the half-open interval $[0, n)$.

Algorithm 1, $\mathrm{RandomPair}(n)$, generates a random pair (i, j) of distinct integers from the interval $[0, n)$, uniformly distributed over the space of all $\binom{n}{2}$ such pairs. Line 1 generates the first of the two integers, i, uniformly from the interval $[0, n)$. Then, to generate the other integer, a random integer j uniformly distributed over the interval $[0, n-1)$ is first generated (line 2). If j is the same as i, we reset j to $n-1$ (lines 3–4), since $n-1$ was excluded via the half-open interval $[0, n-1)$.

Algorithm 2, $\mathrm{RandomTriple}(n)$, generates a random triple (i, j, k) of distinct integers from the interval $[0, n)$, uniformly distributed over the space of all $\binom{n}{3}$ such pairs. 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 handles mapping duplicates to the values excluded by the half-open intervals. The adjustment in lines 4–5 handle the case when i and j are identical in the same way as in Algorithm 1, mapping the duplicate to $n-1$. Lines 6–7 maps k to $n-2$ if it is initially equal to i. It can safely do this without comparing to j since j had just been set to $n-1$. Lines 9–12 handle the case when i and j are already initially different. In that case, we must compare k to both i and j in sequence, mapping k to one of the two values excluded by its half-open interval if it is the same as either i or j. Both comparisons are always necessary since the adjustment on line 10 could potentially create a conflict between k and i that must then be resolved.

The worstcase 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.

Algorithm 2 RandomTriple(n)

```
i ← Rand(n)
j ← Rand(n − 1)
k ← Rand(n − 2)
if j = i then
    j ← n − 1
    if k = i then
        k ← n − 2
else
    if k = j then
        k ← n − 2
    if k = i then
        k ← n − 1
return (i, j, k)
```

Table 1: Important URLs for the $\rho\mu$ open source library

Source	`https://github.com/cicirello/rho-mu`
Website	`https://rho-mu.cicirello.org/`
Maven	`https://central.sonatype.com/artifact/org.cicirello/rho-mu/`

3 Experimental Methodology

We implemented the new algorithms from Section 2, as well as other random sampling algorithms in the open source library $\rho\mu$. See Table 1 for URLs to the libary sourcecode, project website, and in the Maven Central repository. The experiments of this paper specifically use $\rho\mu$ 4.0.0.

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 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 [10] algorithm, which is a faster optimized version of the DotMix [7] algorithm, and which passes the DieHarder [1] 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 [8]. Our prior experiments [3] 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 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) [9] to implement our experiments. 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. We measure and report average time per operation in nanoseconds, along with 99.9% confidence intervals.

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 [2]	sampleInsertion
pool sampling [5]	samplePool
reservoir sampling [11]	sampleReservoir

Table 3: Reproducible results: URLs to experiment code and data

Code	`https://github.com/cicirello/rho-mu/examples`
Data	`https://github.com/cicirello/rho-mu/examples/data/sample-2-and-3`

The code to reproduce the experiments is available on GitHub in a subdirectory of the repository for the $\rho\mu$ library itself, as is the data from my runs of the experiments. Table 3 provides the URLs to the experiment sourcecode and data.

4 Results

4.1 Sampling Pairs of Distinct Integers

This first set of results in Table 4 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 meaningful degree. If it appears that average time is possibly decreasing as n increases for this algorithm, it may be, although it is clearly negligible. As n increases, the probability of the assignment statement on line 4 of Algorithm 1 decreases. The insertion sampling [2] average runtime is likewise approximately constant, though a higher constant (note that the $n = 1024$ case lead to unexpectedly longer average runtime for insertion sampling, and we offer no explanation). Note that insertion sampling's theoretical runtime is $O(k^2)$, which is essentially constant for any fixed k. Pool sampling [5] and reservoir sampling [11] both have an average runtime that increases linearly in n. Reservoir sampling's average runtime is much higher than pool sampling, however, because reservoir 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 is an especially poor choice for low k relative to n.

In the results above, we 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 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: Sampling pairs of distinct integers: average time (nanoseconds) with 99.9% confidence intervals

n	RandomPair (Algorithm 1)	insertion [2]	pool [5]	reservoir [11]
$n = 16$	16.110 ± 0.021	24.651 ± 0.938	55.092 ± 0.294	177.547 ± 0.127
$n = 64$	15.632 ± 0.339	23.838 ± 0.744	89.842 ± 0.196	633.248 ± 5.437
$n = 256$	15.543 ± 0.124	23.678 ± 0.208	300.194 ± 1.691	2081.356 ± 1.574
$n = 1024$	15.460 ± 0.557	31.238 ± 0.133	1209.842 ± 10.175	7931.752 ± 62.058

Table 5: Sampling pairs of distinct integers: average time (nanoseconds) with 99.9% confidence intervals

n	returning new array	preallocated array	Java record
$n = 16$	16.103 ± 0.028	13.754 ± 0.035	12.513 ± 0.259
$n = 64$	15.617 ± 0.061	13.318 ± 0.025	11.983 ± 0.013
$n = 256$	16.056 ± 0.019	13.205 ± 0.027	11.862 ± 0.046
$n = 1024$	15.390 ± 0.052	13.170 ± 0.019	11.812 ± 0.212

The results of this second experiment with sampling pairs of integers is found in Table 5. The fastest of the three options is using a Java record type for the pair of integers. It was even faster than using a preallocated array. Although 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 6 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 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.

In Table 7 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 does not follow the same pattern as that of the pairs of integers case. For triples of distinct integers, the preallocated array case is the fastest, whereas for pairs, the Java record case was the fastest. For triples, the difference in performance between allocating a new array versus a Java record is less clear.

5 Conclusions

In this report, 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 sampling [11] and pool sampling [5] both require linear time, with reservoir sampling being especially computationally expensive for low k since it also requires $n - k$ random integers for sample size k. Although insertion sampling [2] 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.

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

n	RandomTriple (Algorithm 2)	insertion [2]	pool [5]	reservoir [11]
$n = 16$	25.613 ± 0.136	41.957 ± 0.187	58.862 ± 0.214	182.058 ± 1.173
$n = 64$	23.576 ± 1.180	41.965 ± 0.110	91.760 ± 0.762	652.730 ± 1.991
$n = 256$	22.941 ± 0.163	45.928 ± 0.862	287.888 ± 5.217	2097.555 ± 43.353
$n = 1024$	25.234 ± 8.044	41.613 ± 0.242	1177.564 ± 4.234	8582.428 ± 142.582

Table 7: Sampling triples of distinct integers: average time (nanoseconds) with 99.9% confidence intervals

n	returning new array	preallocated array	Java record
$n = 16$	26.145 ± 0.097	21.750 ± 0.073	25.957 ± 0.027
$n = 64$	24.804 ± 0.129	19.878 ± 0.049	22.428 ± 0.013
$n = 256$	23.809 ± 0.022	19.933 ± 0.008	25.339 ± 0.148
$n = 1024$	24.098 ± 0.133	19.481 ± 0.013	21.711 ± 0.742

Our implementations of all of the algorithms of this paper (both new and existing algorithms) are available in the open source Java library $\rho\mu$. The experiments of this paper are also available on GitHub to assist with reproducibility.

References

[1] Robert G Brown, Dirk Eddelbuettel, and David Bauer. *Dieharder: A random number test suite*, 2013. `https://www.phy.duke.edu/~rgb/General/dieharder.php`.

[2] Vincent A. Cicirello. Cycle mutation: Evolving permutations via cycle induction. *Applied Sciences*, 12(11):5506, June 2022. doi:10.3390/app12115506.

[3] Vincent A. Cicirello. $\rho\mu$: A java library of randomization enhancements and other math utilities. *Journal of Open Source Software*, 7(76):4663, August 2022. doi:10.21105/joss.04663.

[4] Vincent A. Cicirello. A survey and analysis of evolutionary operators for permutations. In *Proceedings of the 15th International Joint Conference on Computational Intelligence*, pages 288–299, November 2023. doi:10.5220/0012204900003595.

[5] Jarmo Ernvall and Olli Nevalainen. An algorithm for unbiased random sampling. *The Computer Journal*, 25(1):45–47, 1982. doi:10.1093/comjnl/25.1.45.

[6] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1998.

[7] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–204. ACM, 2012. doi:10.1145/2145816.2145841.

[8] Daniel Lemire. Fast random integer generation in an interval. *ACM Transactions on Modeling and Computer Simulation*, 29(1), 2019. doi:10.1145/3230636.

[9] OpenJDK.org. *Java Microbenchmark Harness (JMH) Version 1.37*, 2023. `https://github.com/openjdk/jmh`.

[10] Guy L. Steele, Jr., Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 453–472. ACM, 2014. doi:10.1145/2660193.2660195.

[11] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11 (1):37–57, 1985. doi:10.1145/3147.3165.