

How to Turn Loaded Dice into Fair Coins

Ari Juels* Markus Jakobsson† Elizabeth Shriver† Bruce K. Hillyer†

June 26, 1998

Abstract

We present a new, optimally efficient technique for extracting unbiased bits from certain physical sources of randomness. Sequences of random numbers are of pervasive importance in cryptography and vital to many other computing applications. While pseudo-random generators are useful for producing such sequences, a physical source of randomness is still needed to seed a pseudo-random generator. Many physical sources of randomness, like radioactive or quantum mechanical sources, possess the useful property of stationarity. In other words, they produce independent outputs on fixed probability distributions over the real numbers. The output of such sources may be viewed as the result of rolling a biased or loaded die. While a loaded die may be a good source of entropy, many applications require input in the form of unbiased random bits, rather than the biased real numbers provided by such dice. Generalizing on a technique described by von Neumann, we present an algorithm for using a loaded die with unknown bias to generate unbiased random bits. We prove our algorithm to be optimally efficient in terms of its output entropy. Experiments conducted using an implementation of the algorithm indicate that the algorithm is efficient and effective in practice. Our algorithm is therefore well suited to taking any stationary physical source and extracting useful randomness in the form of a string of bits.

Keywords: cryptography, pseudo-random, pseudo-random number generator, random number generator, randomness.

*RSA Laboratories, RSA Data Security. E-mail: ari@rsa.com

†Information Sciences Research Ctr., Bell Labs. E-mail: {markusj,shriver,bruce}@research.bell-labs.com

1 Introduction

Countless applications in cryptography, stochastic simulation, search heuristics, and game playing rely on the use of sequences of random numbers. As truly random numbers are a scarce resource, it is common practice to derive such sequences from pseudo-random number generators (PRNGs). A PRNG is an algorithm which takes a truly random input and “stretches” it to produce a long sequence of numbers bearing an appearance of randomness. There is a large body of literature on the design and properties of PRNGs, e.g., [2, 4, 12, 13, 15, 16, 18, 20, 21, 22, 28, 32, 33]. Much less study is devoted, though, to the physical generation and processing of the random input seeds that fuel these PRNGs. It is common practice in the literature to obtain a random seed by invoking a semi-mythical “uniform random source”.

Practitioners have called into service a variety of physical sources of randomness. These include system clocks, radioactive sources [14], quantum mechanical effects in semiconductor devices [1, 25], magnetic disk timings [19], keyboard and mouse timings [26], and lava lamps [8], among others. Timings of human interaction with a keyboard or mouse are currently the most common source of random seeds for cryptographic applications on PCs. After a sufficient amount of such timing data is gathered, it is generally hashed down to a 128-bit or 160-bit seed. This method relies for its security guarantees on unproven or unprovable assumptions about the entropy generated by human users [9] and the robustness of hash functions as entropy extractors.

Some of these sources of randomness, such as radioactive sources and quantum mechanical sources may yield data from probability distributions that are stationary. In other words, the output of these sources does not change over time and does not depend on previous outputs. Even if a source is stationary, though, it generally has a bias. In other words, the source does not give unbiased bits as direct output. Many applications, especially in cryptography, rely on sequences of unbiased bits. It is therefore quite important to be able to extract unbiased bits efficiently from a stationary source with unknown bias.

Suppose that a reading obtained from stationary source of randomness D can be equal to any one of m different values, but that the probability of obtaining any one of these values is unknown. Such a source of randomness can be thought of as a die D with m faces: taking readings from the random source is like rolling the die D . The m faces of D are not necessarily equally probable. In other words, the die D may be loaded. Our aim in this paper is to use a loaded die to simulate fair coin flips in an optimally efficient manner. In a practical setting, this enables us to extract random bits efficiently from a biased physical source of randomness.

1.1 Previous work

Von Neumann [34] offers perhaps the earliest written reference to the problem of simulating unbiased coin flips using a biased coin. He describes the following trick from the folklore. Flip the biased coin twice. If it comes up HT, output an H. If it comes up TH, output a T. Otherwise, start over. This method will simulate the output of an unbiased coin irrespective of the bias of the coin used in the simulation. The tradeoff is this: the stronger the bias of the coin providing input to the simulation, the lower its entropy, and the longer the simulation will take on average to produce output. Exploration of von Neumann’s technique and of various extensions are a favorite classroom exercise, and have also formed the basis of a number of research papers.

One line of research treats generalizations of von Neumann’s method by which it is possible to obtain unbiased coin flips from biased Markov Chains (MCs). These MCs generally assume the following form. Each state in the MC has two outgoing transitions with possibly unequal probabilities. When one of these transitions is followed, the chain outputs a 1; when the other is followed, the chain outputs a 0. A biased die, of course, can be modeled fairly closely by such a MC. In particular, we can construct a binary tree with at least m leaves, where m is the number of sides on the die D we seek to model. If the transitions in the tree are weighted appropriately, then the MC can be constructed such that the probability of reaching a given leaf is equal to the probability of obtaining a given roll on D (with probability 0 for some number of leaves if D is not a power of 2).

Elias [10] is perhaps the first to propose a means of extracting unbiased bits from an MC of this sort. His construction is nearly optimal in the sense that its output entropy approaches that of the underlying MC. The generality of Elias’ algorithm leads to a different type of inefficiency, however. The expected number of steps for which the MC must be run before the first piece of output is obtained is potentially more than the number of states in the chain. Blum [3] proposes an algorithm in which the expected number of steps to obtain output bits is linear in the size of the underlying MC, but the entropy of the output is lower than that in the construction of Elias. Blum’s algorithm is again, however, very general.

In a short note, Dijkstra [7] considers a different extension of von Neumann’s technique. He shows how to use a biased coin (or, alternatively, a biased die) to simulate a fair die with a given number of faces. We can, for instance, simulate a 37-sided die as follows. We flip the biased coin 37 times, and then rotate the resulting sequence to the left until its minimum lexicographic value is achieved. The number of rotations yields the roll of the simulated die. The technique described in our paper involves a consideration of not just rotations, but permutations, and so may be regarded as a natural extension of Dijkstra’s observation.

Related areas of research include that of Feldman et al. [11]. In their exploration of the notion of bounded, rather than expected-time polynomial Turing machines, Feldman et al. seek to find a small set of biased coins capable of efficiently simulating any k -sided die (where k is polynomial in the input length of the Turing machine). They show that for the special case of simulating one n -sided die within $O(\log n)$ coin flips, only two types of biased coin are required – or one, if irrationally biased coins are permitted. Extensions of their work include the papers of Uehara [31] and Itoh [17].

Also of relevance is the work of Santha and Vazirani [27], who show how to use a weak random source of bits to produce bit strings which are not identical to, but indistinguishable from perfectly random ones. Their techniques are based on the simple trick of X-ORing together weakly random sequences. In a follow-up to this work, Boppana and Narayanan [5, 6] examine the expected performance of such weak sources.

1.2 Our result

In this paper, we show how to use a die D with unknown bias to simulate flips of an unbiased coin. As explained above, our result may be regarded as an extension of the technique of von Neumann [34] for simulating unbiased coin flips using a biased coin. Our work is thus similar in spirit to that of Dijkstra [7], but has a different aim. While Dijkstra seeks to use a biased

source to simulate a fair die with a fixed number of sides, we show how to use such a source to simulate a maximal number of fair coin flips. Blum [3] and Elias [10] have already proposed highly efficient constructions to achieve this aim, but their algorithms are extremely general, applying as they do to any form of biased Markov Chain. Our result may thus be regarded as a demonstration that it is possible to do somewhat better for the special case of a biased die. For instance, unlike [10] and [3], our algorithm has no minimal number of observations required before it can produce output. In fact, given any fixed number of rolls of a biased die D , we prove that our algorithm is optimally efficient in the sense that it achieves the maximum possible output entropy. In an asymptotic sense, as we show, our algorithm extracts the full entropy of D . We also show that with careful construction, our algorithm can be implemented so as to require only modest computational resources.

1.3 Organization of the paper

The remainder of this paper is organized as follows. In section 2, we give definitions, and describe our fair bit extractor Q . In section 3, we prove that Q is optimal in terms of output entropy—in particular, that it outputs the maximum possible expected number of bits. We prove in fact in section 4 that the output entropy of Q asymptotically approaches the entropy of the input source; we also give some caveats regarding proper usage of algorithm Q in this section. In section 5 we present experiments conducted with an implemented version of algorithm Q . We conclude in section 6 by suggesting some avenues for further research. A pseudo-code implementation of our proposed fair bit extractor Q is given in the appendix to this paper.

2 Our algorithm

2.1 Definitions

We define a *bit extractor* to be a deterministic algorithm or algorithm Q that does the following. Q takes as input a sequence $X = \{X[1], X[2], \dots, X[n]\}$ of n elements in \mathfrak{R} , where n is any positive integer. Q outputs a sequence b_1, b_2, \dots, b_k of bits, where k is an integer that varies as a function of X . We denote the output of Q on input X by $Q(X)$. Hence Q may be viewed as a function $Q : \mathfrak{R}^n \rightarrow \{0, 1\}^*$. Let D be a fixed probability distribution over a finite set of elements in \mathfrak{R} . As explained above, we may think of D as a die: rolling this die is equivalent to drawing an element from the distribution D . We shall let D^n denote the probability distribution over n rolls of the die D , i.e., the probability distribution from which a sequence X of n rolls is drawn.

We define a *fair bit extractor* to be one which takes inputs from a fixed distribution and outputs bitstrings in which bits are independent and unbiased. In other words, a bit extractor is fair if for any n , any fixed probability distribution D over \mathfrak{R} , and any pair of bitstrings b and b' of equal length, $\Pr_{X \in D^n}[Q(X) = b] = \Pr_{X \in D^n}[Q(X) = b']$. Let $\text{bits}_Q(X)$ denote the number of bits output by extractor Q on input X . A fair bit extractor Q is said to be *optimal* if for any n and D , and all extractors Q' , $\mathbb{E}_{X \in D^n}[\text{bits}_Q(X)] \geq \mathbb{E}_{X \in D^n}[\text{bits}_{Q'}(X)]$. In other words, Q outputs a maximal expected number of bits over all sequences of n readings.

Let $X = \{X[1], X[2], \dots, X[n]\}$ be a sequence of readings drawn from the distribution D^n . We refer to the number of distinct readings in X less than or equal to $X[i]$ as the *rank* of $X[i]$,

denoted by $\text{rank}(X[i])$.

Example 1 Suppose that the die D yields the output sequence $X = \{5, 10, 23, 6, 10, 23\}$. Then $X[1] = 5$, $X[2] = 10$, $X[3] = 23$, $X[4] = 6$, etc. Here, $\text{rank}(X[1]) = 1$, $\text{rank}(X[2]) = 3$, $\text{rank}(X[3]) = 4$, $\text{rank}(X[4]) = 2$, $\text{rank}(X[5]) = 3$, and $\text{rank}(X[6]) = 4$.

We call $\rho(X) = \{\text{rank}(X[1]), \text{rank}(X[2]), \dots, \text{rank}(X[n])\}$ the *rank sequence* of X . We denote by $\{\rho(X)\}$ the *rank multiset* of X . This is the unordered multiset on the set $\rho(X)$. Finally, we let $\text{fr}_X(i)$ denote the number (or frequency) of readings in X with rank i ; $\text{fr}_X(i)$ may also be viewed as the number of occurrences of the integer i in $\rho(X)$ or $\{\rho(X)\}$.

Example 2 Suppose again, as in Example 1, that the die D yields the output sequence $X = \{5, 10, 23, 6, 10, 23\}$. The rank sequence of X is $\rho(X) = \{1, 3, 4, 2, 3, 4\}$. Thus, $\text{fr}_X(1) = 1$, $\text{fr}_X(2) = 1$, $\text{fr}_X(3) = 2$, and $\text{fr}_X(4) = 2$.

2.2 Algorithm description

We shall now describe our algorithm Q for extracting fair bits from a biased die. We shall construct Q as the composition of two algorithms, called Q_1 and Q_2 . The algorithm Q_1 will make use of the biased die D to simulate an unbiased die U . The algorithm Q_2 will take a roll of the simulated unbiased die U and convert it into an unbiased bitstring. Hence, the algorithm $Q = Q_2 \circ Q_1$ will take a sequence of readings X from the biased die D and convert it into an unbiased bitstring.

Algorithm Q_1 : Using biased die D to simulate unbiased die U . The basis of von Neumann's algorithm is following observation. Given that an H and a T occur in successive flips, both orderings of the H and the T are equally likely, i.e., the probability that an HT occurs is equal to the probability that a TH occurs. To construct Q_1 , we shall extend application of this principle from a biased coin to a biased die. Suppose the only information we know is that n rolls of the die D have yielded a set of readings X with the rank multiset $\{\rho(X)\}$. Observe then that any ordering on $\{\rho(X)\}$ is a possible rank sequence $\rho(X)$ for X .

A priori, all such rank sequences $\rho(X)$ are equally probable. E.g., given that $\{\rho(X)\} = \{1, 1, 2, 3\}$, it is equally likely that $\rho(X) = \{3, 2, 1, 1\}$ as it is that $\rho(X) = \{1, 2, 1, 3\}$ as it is that $\rho(X) = \{1, 2, 3, 1\}$, etc.

Using this observation, we can simulate an unbiased die as follows. Given a sequence X , we construct a list of all possible orderings on the multiset $\{\rho(X)\}$ in numerical order, i.e., in ascending order of integer value. Observe that the number of elements in this list is equal to $n! / \prod (\text{fr}_X(i)!)$; let us call this number S . If $\rho(X)$ is the R th element in this list of orderings on $\{\rho(X)\}$, we output R . (In other words, we output the rank of $\rho(X)$ in $\{\rho(X)\}$.) The output R may be seen to represent the result of a single roll of an unbiased die with S sides labeled $1, 2, 3, \dots, S$. We shall call this unbiased die U .

Example 3 Suppose that we obtain the sequence of readings $X = \{10, 5, 15\}$ on three rolls of the die D . This translates into the rank sequence $\rho(X) = \{2, 1, 3\}$. The corresponding rank multiset $\{\rho(X)\} = \{1, 2, 3\}$ has $3! = 6$ possible orderings. In numerical order, these are:

123, 132, 213, 231, 312, 321.

Since the rank sequence $\rho(X)$ comes 3rd in this list, our sequence X corresponds to a roll of $R = 3$ on an unbiased die U with $S = 6$ sides labeled $\{1, 2, \dots, 6\}$.

Algorithm Q_2 : Translating unbiased die U into an unbiased bitstring We now describe the algorithm Q_2 that translates a roll R of the unbiased die U into an unbiased bitstring. The difficulty here lies in resolving the fact that S is not, in general, a power of 2, so that an unbiased mapping from a die roll to bitstrings of a fixed length is not possible. Instead, we begin by partitioning the sides $\{1, 2, \dots, S\}$ of the die U into sets whose sizes are powers of 2. In particular, we partition the sides of U into sets A_1, A_2, \dots, A_j such that the set sizes $|A_1|, |A_2|, \dots, |A_j|$ are unique, decreasing powers of 2. This is equivalent to the following. Let $s_k s_{k-1} \dots s_1 s_0$ be a binary representation of S . Moving from left to right, for each $s_c = 1$, we create a distinct set $A_i \in \{1, 2, \dots, S\}$ such that $|A_i| = 2^c$; thus, j represents the number of bits in $s_k s_{k-1} \dots s_1 s_0$ equal to 1. Although the faces of U may be assigned arbitrarily to the sets $\{A_i\}$, it is convenient to assign them in numerical order, i.e., $1 \in A_1, \dots, S \in A_j$.

To complete the construction of Q_2 , we must assign a mapping from each set A_i to a set of bitstrings. Recall that for each set A_i , $|A_i| = 2^c$ for some c . The algorithm Q_2 , then, maps the elements of A_i to the set $\{0, 1\}^c$ of bitstrings in a one-to-one fashion. This mapping $Q_2 : A_i \rightarrow \{0, 1\}^c$ can be ordered arbitrarily, but it is convenient to make it increasing, i.e., to map larger valued faces of U to larger valued bitstrings. This completes our definition of the algorithm $Q_2 : \{1, 2, \dots, S\} \rightarrow \{0, 1\}^*$. Note that Q_2 will map different sides of U to bitstrings of differing lengths when S is not a power of 2. Note also that if S is odd, then one sequence produces no output, i.e., $c = 0$ for the set A_j .

Example 4 Let us continue our previous example in which we obtained from the die D the sequence of readings $X = \{10, 5, 15\}$. Recall that the algorithm Q_1 mapped this sequence X to a roll of $R = 3$ on an unbiased die U with 6 sides. In the construction of Q_2 , the sides $\{1, 2, 3, 4, 5, 6\}$ of U are partitioned into sets A_1 and A_2 such that $|A_1| = 2^2 = 4$ and $|A_2| = 2^1 = 2$. In particular, $A_1 = \{1, 2, 3, 4\}$ and $A_2 = \{5, 6\}$. The algorithm Q_2 is then defined by the following table.

	R	$Q_2(R)$
A_1 :	1	00
	2	01
	3	10
	4	11
A_2 :	5	0
	6	1

Hence, $Q_2(R) = Q_2(3) = '10'$. In consequence, the bit extractor Q yields as output on $X = \{10, 5, 15\}$ the bitstring $Q(X) = Q_2(Q_1(X)) = '10'$.

We frequently assume in this paper that the die D has a finite number of sides. This assumption simplifies our proofs in the next section. Note, however, that our algorithm Q may

be used equally well for a die D with an infinite number of sides, i.e., a source of randomness which yields any one of an infinite number of values. The proposed efficient implementation of Q which we give in the next subsection is also flexible in this sense.

2.3 Efficient implementation

The longer the sequence X is, the more efficient Q is (as Theorem 2 will suggest). Hence the approach of constructing Q_1 by explicitly listing all possible orderings on $\{\rho(X)\}$ as above is impractical in a real-world setting. This is because, if we use a sequence long enough for Q to yield output efficiently, the list of orderings is likely to be too long. If, for example, there are only 20 readings in the set X , the number of elements in the list in question may be as large as $20!$, which is greater than 2.4×10^{18} .

In a practical implementation, therefore, we must compute $Q_1(X)$ without reference to a table. One way to achieve this is as follows. We examine the readings in X in sequential order, i.e., in step i we examine $X[i]$. Let $\rho_i(X)$ denote the rank set on the truncated sequence $\{X[i], X[i+1], \dots, X[n]\}$, and let $\text{rank}_i(X[i])$ denote the rank of $X[i]$ in this truncated sequence. *A priori* – i.e., given knowledge of $\rho_i(X)$, but not of $\text{rank}_i(X[i])$ – the value $\text{rank}_i(X[i])$ will be equal to any of the elements in $\rho_i(X)$ with equal probability. Hence, *a posteriori*, we may regard $\text{rank}_i(X[i])$ as the result of rolling an unbiased die U_i whose faces consist of the set $\rho_i(X)$. This means that U_i is a die with $n - i + 1$ faces, all of which are equally likely. Note, however, that U_i may have multiple faces with the same label, as $\rho_i(X)$ may contain repeats.

Example 5 Suppose that $X = \{15, 10, 5, 15, 5\}$. Let us consider the die U_3 . The rank set $\rho(X) = \{3, 2, 1, 3, 1\}$. The rank set $\rho_3(X) = \{\text{rank}_3(5), \text{rank}_3(15), \text{rank}_3(5)\} = \{1, 2, 1\}$. This means that *a priori*, the die U_3 has three faces—namely those in the set $\{1, 2, 1\}$. The *a priori* probability of obtaining a 1 on this die is $2/3$, while that of obtaining a 2 is $1/3$. Since $\text{rank}_3(X[3]) = \text{rank}_3(5) = 1$, *a posteriori*, the actual roll obtained on U_3 is a 1.

In an algorithm to compute $Q_1(X)$, i.e., the roll value R on the full, unbiased die U , we compute the results of the individual rolls U_1, U_2, \dots, U_n and combine them cumulatively into a single die roll. Details on how to achieve this, with accompanying pseudo-code, are available in Appendix A. Computing Q_2 in an efficient manner is somewhat more straightforward. Details and pseudo-code are, again, available in Appendix A.

3 Proof of optimality

Our aim in this section is to prove that Q is an optimal fair bit extractor.

3.1 Preliminaries

Let us assume that $n > 1$, i.e., that X is a non-trivial sequence. Let us also assume, without loss of generality, that the die D has sides labeled $1, 2, 3, \dots, m$. Thus, D has m faces. Let $D(i)$ be the probability that when the die D is rolled, it comes up on face labeled i . Recall that we denote by D^n the distribution on n rolls of the die D . Let $D^n(X)$ therefore denote the

probability that D yields the sequence X on n rolls, i.e., that D yields $X[1]$ on the first roll, $X[2]$ on the second roll, etc. Observe that $D^n(X) = D(X[1]) \times D(X[2]) \times \cdots \times D(X[n])$.

Definition 1 Let $X = \{X[1], X[2], \dots, X[n]\}$ be a sequence of length n . We shall refer to the set of all distinct permutations of X as the permutation class of X , which we shall denote by $\Pi(X)$.

If, for example, $X = \{1, 3, 1\}$, then $\Pi(X) = \{\{1, 1, 3\}, \{1, 3, 1\}, \{3, 1, 1\}\}$. Observe that for any $X_1, X_2 \in \Pi(X)$, it is the case that $D^n(X_1) = D^n(X_2)$. In other words, we can state the following observation.

Observation 1 All sequences in (elements of) a given permutation class $\Pi(X)$ are equally probable.

The following definition is central to our proofs.

Definition 2 Let Q be a bit extractor, and $\Pi(X)$ be a permutation class. Q is said to be uniform over $\Pi(X)$ if for all pairs of bitstrings b and b' of equal length, Q maps the same number of sequences from $\Pi(X)$ to b as it does from $\Pi(X)$ to b' . More formally, Q is uniform over $\Pi(X)$ if $|\{X_1 \in \Pi(X) \text{ s.t. } Q(X_1) = b\}| = |\{X_2 \in \Pi(X) \text{ s.t. } Q(X_2) = b'\}|$. A bit extractor Q is said to be uniform if it is uniform over all permutation classes $\Pi(X)$.

3.2 Proof sketch

Recall that our goal is to prove that Q is an optimal fair bit extractor. The first step in our proof will be to obtain a more manageable notion of fairness. In particular, we shall show that a bit extractor V is fair if and only if it is uniform. To do so, we shall construct a special die D for which the weighting of the sides drops off very rapidly: D will be such that $D(1) \gg D(2) \gg \cdots \gg D(m)$. This die D will induce a separation of permutation classes by probability. In particular, as we shall show in Lemma 1, if sequences X and Y do not belong to the same permutation class, then the probability of obtaining X in n rolls of the die D must be very different from the probability of obtaining Y in n rolls of the die D . This separation of permutation classes will aid in our proof of Lemma 2. Lemma 2 states that in order to be fair, an extractor V must be uniform. We shall in fact prove the lemma by contradiction. We shall suppose that V is not uniform, and consequently biased on some permutation class $\Pi(X)$. Because of the separation of permutation classes induced by die D , then, it will be impossible for V to be fair. In particular, V will not be able compensate for its bias over $\Pi(X)$ with a bias over other permutation classes. In Corollary 2, we extend the result in Lemma 2 to show that a bit extractor is fair if and only if it is uniform. To prove that Q is an optimal fair bit extractor, then, we shall only need to show that Q is optimal among bit extractors that are uniform. This is what we do in Theorem 1, our main theorem.

3.3 Proof details

Recall that our first goal is to show that if a bit extractor V is fair, then it is uniform. We begin by constructing a heavily weighted die D in which $D(1) \gg D(2) \gg \dots \gg D(m)$. In particular, for all values i , we shall let $D(i) \propto 1/h^{n^i}$, where $h = m^n$. (In other words, $D(i) = w/h^{n^i}$ for some normalizing factor w that makes $\sum D(i) = 1$.) Lemma 1 and Corollary 1 show how such a die D induces large differences in the probabilities on sequences from different permutation classes.

Lemma 1 *If X and Y are not in the same permutation class and $D^n(X) \geq D^n(Y)$, then $D^n(X)/D^n(Y) \geq m^n$.*

Proof: By definition of D , $D^n(X) \propto h^{-p(n)}$ for a polynomial $p(z)$ of the following form: $p(z) = a_m z^m + a_{m-1} z^{m-1} + \dots + a_1 z$, where a_i is the number of occurrences of roll i in the sequence X . Similarly, $D^n(Y) \propto h^{-q(n)}$ for a polynomial $q(z) = b_m z^m + b_{m-1} z^{m-1} + \dots + b_1 z$, where b_i is the number of occurrences of roll i in the sequence Y . Note that $\sum a_i = \sum b_i = n$.

It follows that $D^n(X)/D^n(Y) = h^{q(n)-p(n)}$, where $q(z) - p(z) = (b_m - a_m)z^m + (b_{m-1} - a_{m-1})z^{m-1} + \dots + (b_1 - a_1)z$. Since X and Y are not in the same permutation class, $q(z) \neq p(z)$. Since $D^n(X) \geq D^n(Y)$, it follows that the first (i.e., leftmost) non-zero coefficient of $q(z) - p(z)$ is positive. Since $\sum a_i = \sum b_i = n$, it is easy to see that the value of $q(n) - p(n)$ is minimized for positive n when $(b_m - a_m) = 1$, $(b_{m-1} - a_{m-1}) = -n$, and $(b_1 - a_1) = n - 1$. When this is the case, $q(n) - p(n) \geq 1$ for $n > 1$. Hence $D^n(X)/D^n(Y) \geq h = m^n$. ■

Corollary 1 *Let X be a sequence of length n , and let Y_1, Y_2, \dots, Y_k be all sequences of length n such that $D^n(Y_i) < D^n(X)$. Then $\sum D^n(Y_i) < D^n(X)$.*

Proof: Since D has m faces, the number of possible sequences of length n is at most m^n . Hence the number of sequences Y_i such that $D^n(Y_i) < D^n(X)$ is less than m^n , i.e., $k < m^n$. By Lemma 1, $D^n(X)/D^n(Y_i) \geq m^n$. The corollary follows. ■

We are now ready to show the following.

Lemma 2 *If V is a fair bit extractor, then V is uniform.*

Proof: We shall prove this by contradiction. Suppose V is a fair bit extractor that is not uniform. Then there exists a permutation class over which V is not uniform. Let us assume, w.l.o.g., that among permutation classes $\Pi(Y)$ over which V is non-uniform, $\Pi(X)$ is such that $D^n(Y)$ is maximal. Note that by Lemma 1, $\Pi(X)$ is uniquely defined.

Since V is non-uniform over $\Pi(X)$, there exist bitstrings b and b' of equal length such that V maps more elements of $\Pi(X)$ to b than to b' . Since V is fair, V must output b and b' with equal probability over sequences taken from die D . Let \mathcal{G} be the set of sequences that V maps to b , and let \mathcal{H} be the set of sequences that V maps to b' . Let $D^n(\mathcal{G})$ denote $\sum_{Y \in \mathcal{G}} D^n(Y)$, i.e., the probability of obtaining a sequence in \mathcal{G} or, equivalently, the probability of outputting bitstring b . Let $D^n(\mathcal{H})$ be defined similarly. The fairness of V , then, means that $D^n(\mathcal{G}) = D^n(\mathcal{H})$. By our assumption on the non-uniformity of V , however, there are more elements from $\Pi(X)$ in \mathcal{G} than in \mathcal{H} .

Let us now remove from both \mathcal{G} and \mathcal{H} all sequences Y such that $D^n(Y) > D^n(X)$. By the maximality of $D^n(X)$ among permutation classes over which V is non-uniform, it must be the case for any such Y that V is uniform over the permutation class $\Pi(Y)$. Therefore, after removing all such Y from \mathcal{G} and \mathcal{H} , it will still hold true that $D^n(\mathcal{G}) = D^n(\mathcal{H})$. Let us now remove from \mathcal{H} all elements of $\Pi(X)$. Since there were more elements of $\Pi(X)$ in \mathcal{G} than in \mathcal{H} , we can also remove the same number of elements of $\Pi(X)$ from \mathcal{G} , and still have at least one $X' \in \mathcal{G}$ such that $X' \in \Pi(X)$. By Observation 1, D^n is constant across all elements of the same permutation class, so it is still the case that $D^n(\mathcal{G}) = D^n(\mathcal{H})$. It follows that $D^n(\mathcal{H}) \geq D^n(X')$. Since the only sequences Y now remaining in \mathcal{H} are those such that $D^n(Y) < D^n(X)$, this is a contradiction of Lemma 2. ■

Corollary 2 *V is a fair bit extractor if and only if V is uniform.*

Proof: By definition, if V is uniform, then it maps elements from any permutation class evenly among all bitstrings of a given length. Recall also that for any $X_1, X_2 \in \Pi(X)$, it is the case that $D(X_1) = D(X_2)$. If we combine these two facts, we immediately see that if V is uniform, then V is a fair bit extractor. This, in conjunction with Lemma 2, proves the corollary. ■

Theorem 1 *The algorithm Q is an optimal fair bit extractor.*

Proof: By Corollary 2, a bit extractor V is fair if and only if it is uniform, i.e., if and only if it maps all elements of a permutation class $\Pi(X)$ evenly across bitstrings of any given length. This means that V is a fair bit extractor if and only if $\Pi(X)$ can be partitioned into sets A_1, A_2, \dots, A_j such that for any i , $|A_i| = 2^c$ for some c , and $V : A_i \rightarrow \{0, 1\}^c$ is a one-to-one mapping.

Let $\text{bits}_V(\Pi(X))$ denote the expected number of bits output using V over sequences from the permutation class $\Pi(X)$, as calculated over the distribution D^n . Observe that $\log_2 |A_i| = c$ is the number of bits V outputs when given an input from A_i . Observe too that if $X' \in \Pi(X)$, then $\Pr[X' \in A_i] = |A_i|/|\Pi(X)|$. Let $h = 1/|\Pi(X)|$. It is easy to see, then, that $\text{bits}_V(\Pi(X)) = h \sum |A_i| \log_2 |A_i|$.

Suppose that a set A_i is bisected into new sets A'_i and A''_i (and V is modified appropriately to preserve uniformity). Let Δ_{bits} denote the consequent change in $\text{bits}_V(\Pi(X))$. By the calculations above, it is straightforward to show that $\Delta_{\text{bits}} = 2h|A_i|(\log_2 |A_i|/2) - h|A_i| \log_2 |A_i| = -h|A_i|$. Hence, bisecting a set A_i causes $\text{bits}_V(\Pi(X))$ to decrease. On the other hand, let us suppose that there exist sets A_h and A_i such that $|A_h| = |A_i|$, and that these sets are merged (and V is modified appropriately). In this case, conversely, $\Delta_{\text{bits}} = +h|A_i|$. Thus, this merging of equal-sized sets causes $\text{bits}_V(\Pi(X))$ to increase. It follows that $\text{bits}_V(\Pi(X))$ is maximized when no merging of sets is possible, in other words, when the sizes of the A_1, A_2, \dots, A_j are distinct. As our proposed bit extractor Q partitions all permutation classes $\Pi(X)$ into such distinct sets, Q maximizes $\text{bits}_Q(\Pi(X))$ for all permutation classes $\Pi(X)$.

By linearity of expectation, since Q maximizes $\text{bits}_Q(\Pi(X))$ for all permutation classes $\Pi(X)$, the expected number of bits output by Q is maximal. This is to say that $E_{X \in D^n}[\text{bits}_Q(X)]$ is maximal over all fair bit extractors. Therefore Q is optimal. ■

Note that our proof assumes a die D with a finite number of sides m . It is possible to generalize our proof to dice with infinite number of sides, i.e., to random sources with infinite numbers of possible outputs. For the sake of simplicity, we omit this more general proof.

4 Output efficiency and proper usage

In this section, we provide further analysis of our algorithm Q . First, we shall prove that our algorithm is optimal in another sense: the number of output bits it yields is asymptotic to the Shannon entropy of the input source. For the sake of simplicity, we shall restrict our proof to the case in which the random source is a biased coin, but the proof is extensible to the case of a biased die. Then we shall give some caveats about proper use of the algorithm Q .

4.1 Proof of asymptotic output efficiency

4.1.1 Preliminaries

Our algorithm Q generalizes von Neumann’s technique to simulations involving real-valued probability distributions. It is well known, however, that even for simulations that involving biased coins, there are algorithms more efficient than von Neumann’s [24]. Our algorithm is among these—in fact, as we have just proven, it is optimally efficient. Consider a biased coin yielding H with probability $3/4$ and T with probability $1/4$. If we flip such a coin four times, our proposed extraction algorithm will yield $165/128 = 1.29$ bits of output on average, while von Neumann’s technique will yield only 0.75 bits. This is because von Neumann’s technique does not make use of all information available in the simulation. If the biased coin yields TTHH, for instance, von Neumann’s technique will produce no output, even though the coin flip sequence may be regarded as containing information.

It is possible to characterize the full amount of randomness inherent in a random source. The most common measure is known as the Shannon entropy [29, 30], and is defined as follows.

Definition 3 *Let D be a probability distribution over \mathbf{Z}^+ . Then the number of bits of Shannon entropy of D , denoted by $SH(D)$, is defined to be $-\sum_{i=1}^{\infty} D(i) \log_2 D(i)$, where, by convention, $D(i) \log_2 D(i) = 0$ if $D(i) = 0$.*

The Shannon entropy serves as an absolute reference point by which we may gauge the output efficiency of a bit extractor. We noted that von Neumann’s algorithm is not optimally efficient. Let C denote the output of a coin with constant bias p , and C^n denote the distribution on n flips of the coin C . Let $vN(C^n)$ be the expected number of output bits yielded by von Neumann’s algorithm on the distribution C^n . The following fact quantifies the inefficiency of von Neumann’s algorithm.

Fact 1 *For an unbiased coin C , $vN(C^n)/SH(C^n) \leq 1/4$.*

The algorithm of von Neumann achieves its maximal efficiency when applied to an unbiased coin. In this case (assuming that n is even), $vN(C^n)/SH(C^n) = 1/4$. As we shall show, however, the algorithm Q presented in this paper extracts nearly all of the Shannon entropy of a random source. In the interest of simplifying our proof, we shall show this only for the case in which entropy is being extracted from a biased coin. The result applies equally well, however, to any die D with a constant number of sides.

4.1.2 Proof sketch

We give our proof in two parts. Recall that Q_2 takes as input the roll of an unbiased die U with S sides and outputs unbiased bitstrings. It is easy to see that $SH(U) = \log_2 S$. In Lemma 3, we show that Q_2 is efficient. In particular, we show that on average, Q_2 extracts at least $\lfloor \log_2 S \rfloor - 1$ bits from a roll of the die U —very nearly the full Shannon entropy of U . Recall now that Q_1 takes a sequence of readings (coin flips in our proof), and outputs the roll of a simulated, unbiased die U with S faces. In Theorem 2, we effectively show that Q_1 is efficient. The entropy of the input to Q_1 may be viewed as the entropy associated with the choice of permutation class Π plus the entropy associated with the roll of an unbiased die U . The entropy of the output of Q_1 consists of the entropy generated by the die U . We show that the entropy associated with U is generally much larger than the entropy associated with the choice of Π . In other words, we prove that on average,

Q_1 yields a die U with a large number S of faces—large enough so that the entropy of U approaches the entropy of the random source D^n . Together with Lemma 3, this demonstrates that our algorithm as a whole asymptotically extracts the full entropy of D^n .

4.1.3 Proof

Lemma 3 *If the unbiased die U has S sides, then Q_2 will extract at least $\lfloor \log_2 S \rfloor - 1$ bits on average from a roll of U .*

Proof: Suppose that $s_j s_{j-1} \dots s_0$ is a binary representation of S such that $s_j = 1$. Let $B(S)$ expected number of bits output by Q_2 on an unbiased die U with S faces. It is easy to see by definition of Q_2 that $B(S) = (1/S) \sum_{i=1}^j i 2^{s_i}$.

Suppose $S' = 2^{j+1} - 1$, i.e, S' is such that $s'_j = s'_{j-1} = s'_{j-2} = \dots = s'_0 = 1$. Observe by straightforward calculation using the definition of B that $B(S') > j - 1$. Let us suppose then that we flip a set of ‘1’ bits in S' to ‘0’ bits. In particular, let us suppose that we flip a series of k distinct bits, and that the last of these bits is s'_i . We shall call the resulting integer S'' . Let us now prove by induction that $B(S'') \geq B(S')$. If $k = 0$, then $S'' = S'$, so $B(S'') = B(S')$. Suppose now that $k > 0$, and that S'' is the integer obtained by executing on S' all ‘1’ to ‘0’ flips except the flip on s'_i . By induction, $B(S'') > j - 1$. By definition of B , we see that $B(S''') = (S'' B(S'') + i 2^i) / (S'' + 2^i)$. By straightforward algebra on this last expression combined with the fact that $B(S'') > j - 1$, we are able to show that $B(S''') \geq B(S'')$. This completes the inductive proof. Since we showed that $B(S''') \geq B(S')$, and started with $B(S') > j - 1$, it follows that for any integer S , it will be the case that $B(S) > j - 1$. This proves the lemma. \blacksquare

For the proof of our theorem, let $\text{bits}_Q(D^n)$ be the expected number of bits output by Q over inputs from D^n and let $\text{poly}(n)$ denote a quantity which is polynomial in n . We use the symbol \sim to denote equality asymptotic in n . In other words, $a \sim b$ if $\lim_{n \rightarrow \infty} a/b = 1$.

Theorem 2 *For any biased die D , $\text{bits}_Q(D^n) \sim SH(D^n)$.*

Proof: Observe that a permutation class $\Pi(X)$ and the output $Q_1(X)$ are together sufficient to determine X uniquely. Hence, $SH(\Pi(D^n)) + SH(Q_1(D^n)) \geq SH(D^n)$. By Lemma 3,

$\text{bits}_Q(D^n) = SH(Q_2(Q_1(D^n))) \geq SH(Q_1(D^n)) - 2$. Combining this equation with the previous equation yields $SH(\text{bits}_Q(D^n)) \geq SH(D^n) - SH(\Pi(D^n)) - 2$.

Let k here represent the number of possible outcomes of the die D . A permutation class may be uniquely specified by the number of appearances of these k outcomes in a sequence of n rolls. It follows that $|\Pi(D^n)| \leq n^k$, and hence that $SH(\Pi(D^n)) \leq \log_2(n^k) = k \log_2 n$. Plugging this into the expression for $\text{bits}_Q(D^n)$ above reveals that $\text{bits}_Q(D^n) \geq SH(D^n) - k \log_2 n - 2$. Since it is obviously also the case that $\text{bits}_Q(D^n) \leq SH(D^n)$, the theorem follows. ■

Theorem 2 shows that the expected number of bits output by Q on any biased die D asymptotically approaches the full entropy of the die D . The larger the number n of rolls of D input to Q , the closer Q is to approaching full efficiency.

4.2 Proper usage of algorithm Q

Although Q outputs unconditionally unbiased bits when applied correctly, it is important to make careful use of the bit extractor Q . As the following example shows, there are uses of Q that may seem correct on first inspection, but in fact yield biased output.

Example 6 Suppose a user interested in obtaining a single unbiased bit from a die D makes use of the following algorithm. Take readings $X[1], X[2], \dots$ from D until a sequence X is obtained such that $Q(X)$ consists of at least one bit. Then output $Q(X)$ and halt.

It may seem surprising, but this algorithm yields biased output. Let us suppose that D is really a coin, and has two sides numbered 1 and 2. It is easy to see that the described algorithm will halt on any sequence of the form $\{1, 1, 1, \dots, 1, 2\}$ and output a ‘0’, and halt on any sequence of the form $\{2, 2, 2, \dots, 2, 1\}$ and output a ‘1’. The algorithm will not halt on any other sequence. In particular, no output will be yielded on sequences of the form $\{1, 1, 1, \dots\}$ or $\{2, 2, 2, \dots\}$. If $D(1) = p$ and $D(2) = 1 - p$, it is easy to see that the probability that the described algorithm outputs the bit ‘0’ is p , while the probability of a ‘1’ output is $1 - p$. Hence this use of Q yields *biased* output.

In order to ensure that the output of the extractor Q is unbiased, the user should provide Q with a sequence X of fixed length or of length independent of the properties of readings in X .

5 Implementation and experiments

In this section we describe an implementation of algorithm Q , and experimental results concerning the effectiveness of the algorithm in practice.

Pseudo-code for algorithm Q is given in the appendix. We made a straightforward translation of this pseudo-code to the C programming language, and used the GNU Multiple Precision Arithmetic Library Edition 2.0.2 (June 1996)¹ for arithmetic involving factorials and the three variables in the pseudo-code (namely, S, L, and F) that may contain large values.

The purpose of implementing the code was to investigate whether the algorithm is as effective in practice as the theorems suggest that it should be. In particular, we wish to know the speed of

¹GNU MP source and documentation is available via <http://www.fsf.org/order/ftp.html>

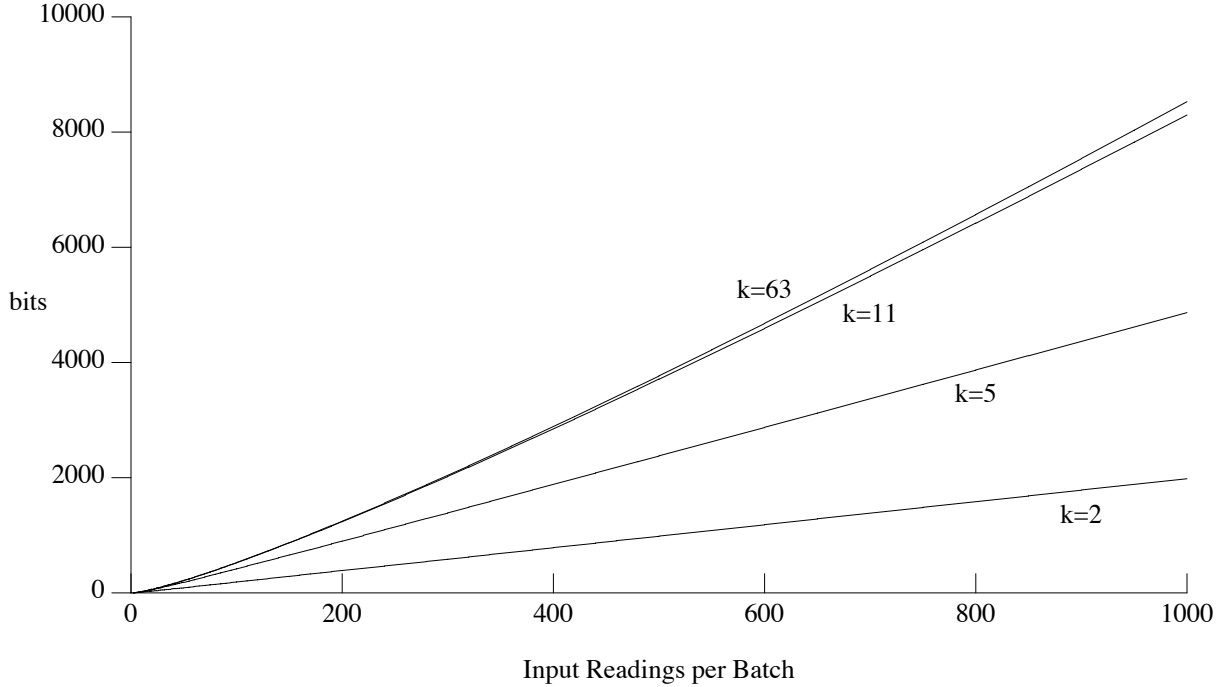


Figure 1: Average Output Bits Produced per Input Batch.

the algorithm in practice, and we wish to know how many output bits are generated from a given number of input bits of known bias. The timing experiments were conducted on a lightly-loaded Sun Ultra-1 computer running the Solaris 2.6 operating system.

A *reading* is a k -bit integer, where k is a parameter of the experiment. We used values of k from 2 to 63. Each reading is constructed one bit at a time, with the same specified bias applied independently to each bit. Each bit is generated using one call to the C library builtin function `lrand48()`, which is a standard linear congruential pseudo-random generator that uses 48-bit arithmetic internally, and that returns a uniformly-distributed 32-bit integer. We generate a bit that is 1 with probability p by seeing whether `lrand48() mod 1/p` is zero. The values of p used in the experiments are $1/2$, $1/4$, $1/8$, and $1/16$. A *batch* is a sequence of b readings given as an input to one invocation of the functions Q_1 and Q_2 . We experimented with values of b ranging from 2 to 1000. An experimental *trial* first generates an input batch according to parameters k , b , and p , and then executes Q_1 and Q_2 on that batch. Each experiment runs 100 trials for each setting of the parameter triple $\{k, b, p\}$.

The first experimental results involve the speed of the algorithm. The graph of figure 1 shows the average number of output bits produced by each call to Q . We observe that the number of output bits increases approximately linearly with increasing batch size b , and the slope of the line is determined by the bits per reading k . For instance, with $k = 11$ and $p = 1/2$ (i.e., unbiased input), we observe that as the batch size b increases from 2 to 1000, the average number of output bits per batch increases from 1 to 27,071. But larger values of b cause the

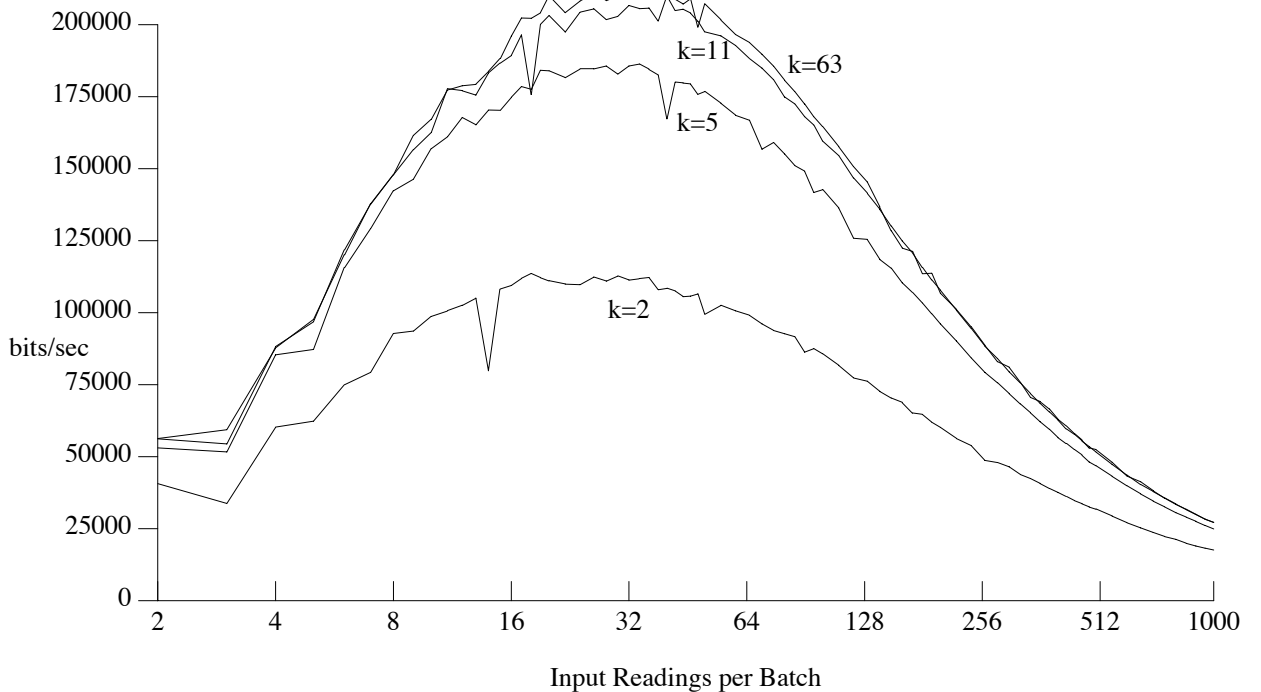


Figure 2: Average Output Bits Per Second Produced.

algorithm to compute with much larger multi-precision values, increasing the run time. For our implementation running on an Sun Ultra-1 computer, we observe that as b increases from 2 to 1000, the average time to run the algorithm on one batch of inputs increases from 2 milliseconds to 300 milliseconds. As a consequence of the interplay between increasing output bits per batch and increasing time per batch, the rate of output bit generation first increases, then decreases. We see this clearly in the graph of figure 2, which shows the average number of output bits per second produced by our implementation running on an Sun Ultra-1 computer, when the input bits are unbiased. The rate of production reaches a peak of about 210,000 bits per second at $k = 11$ and $b = 40$. We ran our experiments in the background on a timeshared machine. As a consequence, the experimental timings were occasionally perturbed by other activity on the system. This interference is seen in the deep notches and other irregularities in figure 2.

The second experiment explores how effective the algorithm is in extracting all the randomness available in the input. Let $n_{k,b,p}$ denote the average number of output bits when the input has k bits per reading, b readings per batch, and each input bit is 1 with probability p . Then we define the *simple efficiency* of the algorithm to be $n_{k,b,p}/(k \times b)$, i.e., the ratio of output bits to input bits.

The number of bits of randomness available in an input batch is given by the Shannon entropy (recall definition 3). In particular, the probability p determines the entropy per bit E_p . For the tested values of $p = 1/2, 1/4, 1/8, 1/16$ the corresponding bit entropies are approximately 1, 0.81, 0.54, and 0.34 respectively. The number of bits of randomness in an input reading is $E_p \times k$,

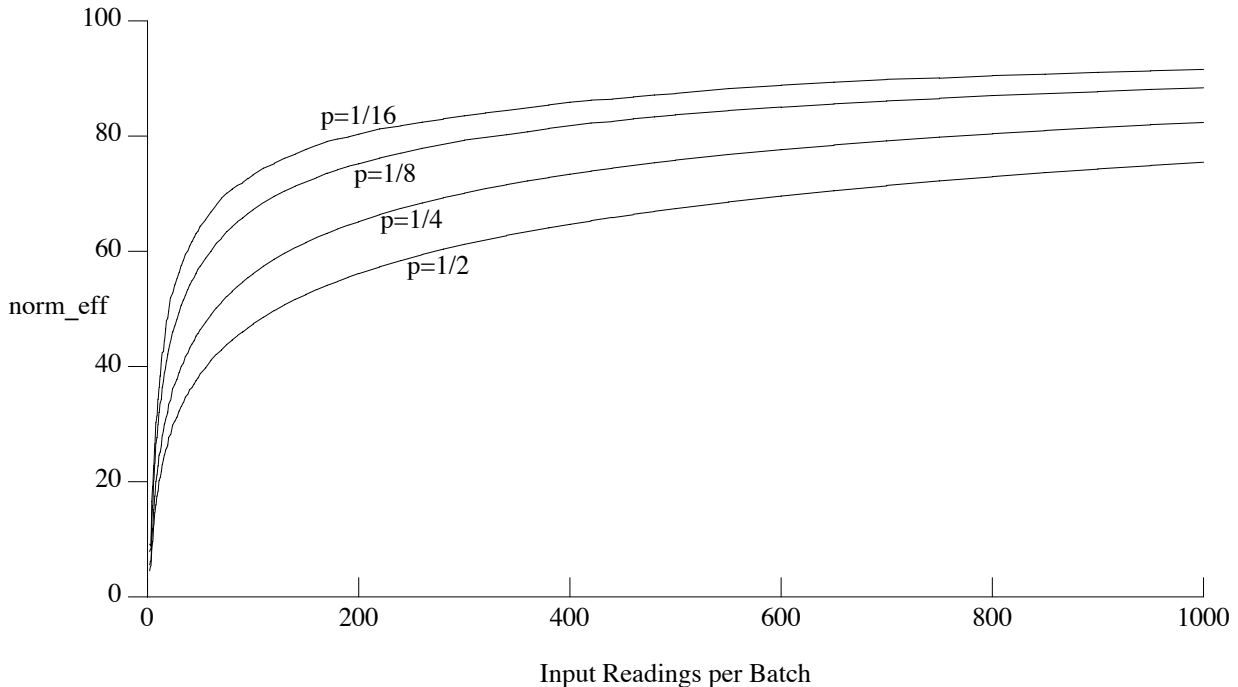


Figure 3: Normalized Efficiency as a Function of Batch Size.

and thus the number of bits of randomness in an input batch is $E_p \times k \times b$. As a consequence, we define the *normalized efficiency* of the algorithm to be $n_{k,b,p}/(E_p \times k \times b)$. Theorem 2 tells us that the normalized efficiency approaches 1 as b increases. The graph in figure 3 illustrates the speed at which this convergence occurs in practice. We observe dramatic increases of efficiency up to about $b = 100$, and slower increases for batch sizes larger than that. We also observe that the efficiency increases with increasing bias in the input. With unbiased input, the normalized efficiency is 75% for $k = 11$ and $b = 1000$. This increases to 92% when $p = 1/16$.

6 Conclusions and open problems

We have presented an algorithm Q for extracting unbiased bits from a biased die. Q is proven to be optimally efficient in terms of its output entropy, and experimental evidence demonstrates that Q is effective in practice.

Yet a number of open problems remain. The algorithm Q takes a sequence X of n readings from an die D with unknown bias and extracts unbiased bits from X . What if the source D is not really a die, though? In other words, what if there are possible correlations between rolls of D ? In this case, there is no way to obtain bits that are guaranteed to be completely independent and unbiased. Santha and Vazirani [27] propose a simple method of taking correlated coin flips and obtaining bits that are computationally independent and unbiased, i.e., indistinguishable from truly random ones. The following question naturally arises: by analogy with the algorithm

of von Neumann, can the algorithm of Santha and Vazirani be efficiently extended from the case of correlated coin flips to the case of correlated die rolls?

Algorithm Q is designed to work when the bias of die D is unknown. What if the bias is known, though? In this case, is there an extraction algorithm more efficient than algorithm Q ? The authors conjecture that in most cases, even if the bias of D is known, Q is optimally efficient. A simple proof of this fact might be edifying.

Acknowledgments. The authors wish to thank Burt Kaliski for his helpful comments on drafts of this paper, and for proposing a simplification to the pseudo-code implementation of our algorithm. We also wish to thank Daniel Bleichenbacher for proposing a simplification and extension of the proof of Theorem 2 in this paper.

References

- [1] G.B. Agnew. Random sources for cryptographic systems. In *Advances in Cryptology—Eurocrypt '87*, pages 77–81. Springer-Verlag, 1988.
- [2] L. Blum, M. Blum, and M. Shub. A simple, unpredictable pseudo-random generator. *SIAM Journal on Computing*, 15(2):364–383, 1986.
- [3] M. Blum. Independent unbiased coin flips from a correlated biased source: a finite state Markov chain. In *25th IEEE Symposium on Foundations of Computer Science*, pages 425–433, 1984.
- [4] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13:850–864, 1984. A preliminary version appears in FOCS, 1982, pages 112–117.
- [5] R.B. Boppana and B.O. Narayanan. The biased coin problem. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 252–257, 1993.
- [6] R.B. Boppana and B.O. Narayanan. The biased coin problem. In *SIDISC*, 9(1):29–36, 1996. Preliminary version in *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*.
- [7] E. Dijkstra. Making a fair roulette from a possibly biased coin. *Information Processing Letters*, 36(4):193, 1990.
- [8] Computing random numbers. Light headed. *The Economist*, 31 May 1997.
- [9] D. Eastlake, S. Crocker, and J. Schiller. RFC1750: Randomness recommendations for security, Dec 1994.
- [10] P. Elias. The efficient construction of an unbiased random sequence. *Ann. Math. Statist.*, 43(3):865–870, 1972.
- [11] D. Feldman, R. Impagliazzo, M. Naor, N. Nisan, S. Rudich, and A. Shamir. On dice and coins: Models of computation for random generation. *Information and Computation*, 104(2):159–174, June 1993.
- [12] O. Goldreich. A note on computational indistinguishability. Technical Report TR-89-051, International Computer Science Institute, Berkeley, CA, July 1989.
- [13] O. Goldreich, H. Krawczyk, and M. Luby. On the existence of pseudorandom generators. *SIAM Journal on Computing*, 22(6):1163–1175, December 1993. A preliminary version appears in FOCS, 1988, pages 12–24.
- [14] M. Gude. Concept for a high-performance random number generator based on physical random phenomena. *Frequenz*, 39:187–190, 1985.
- [15] J. Håstad. Pseudo-random generators under uniform assumptions. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 395–404, Baltimore, MD, 1990.
- [16] R. Impagliazzo, L. Levin, and M. Luby. Pseudo-random generation from one-way functions. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 12–24, 1989.
- [17] T. Itoh. Simulating Fair Dice with Biased Coins. *Information and Computation*, 126(1):78–82, 1996.
- [18] R. Impagliazzo and D. Zuckerman. How to recycle random bits. In *30th IEEE Symposium on Foundations of Computer Science*, pages 248–253, 1989.

- [19] M. Jakobsson, E. Shriver, B. Hillyer, and A. Juels. A practical secure physical random bit generator. To appear in *Proceedings of the 5th ACM Conference on Computer and Communications Security*, 1998.
- [20] B. Kaliski. A pseudo-random bit generator based on elliptic curves. In *Advances in Cryptology—Crypto '86*, pages 84–103. Springer-Verlag, 1986.
- [21] L. Levin. One-way functions and pseudorandom generators. *Combinatorica*, 7(4):357–363, 1987. A preliminary version appears in STOC, 1985, pages 363–365.
- [22] M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, New Jersey, 1996.
- [23] C. McDiarmid. *Survey in Combinatorics*, chapter On the Method of Bounded Differences. Cambridge University Press, 1989.
- [24] R. Motwani and P. Raghavan. *Randomized Algorithms*, page 25. Cambridge University Press, 1995.
- [25] M. Richter. *Ein Rauschgenerator zur Gewinnung von Quasi-idealen Zufallszahlen für die Stochastische Simulation*. PhD thesis, Aachen University of Technology, 1992. In German.
- [26] RSA Data Security, Inc. *RSA SecurPC for Windows 95 Users Manual*, 1997.
- [27] M. Santha and U. Vazirani. Generating quasi-random sequences from slightly-random sources. In *25th IEEE Symposium on Foundations of Computer Science*, pages 434–440, 1984.
- [28] A. Shamir. On the generation of cryptographically strong pseudo-random sequences. *ACM Transactions on Computer Systems*, 1(1):38–44, 1983. A preliminary version appears in the 8th ICALP, 1981, pages 544–550.
- [29] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(4):379–423, 623–656, 1948.
- [30] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.
- [31] R. Uehara. Efficient simulations by a biased coin. *Information Processing Letters*. 56(5): 245–248, 1995.
- [32] U.V. Vazirani and V.V. Vazirani. Efficient and secure pseudo-random number generation. In *25th IEEE Symposium on Foundations of Computer Science*, pages 458–463, 1984.
- [33] U.V. Vazirani and V.V. Vazirani. Efficient and secure pseudo-random number generation. In *Advances in Cryptology—Crypto '84*, pages 193–202. Springer-Verlag, 1985.
- [34] J. von Neumann. Various techniques used in connection with random digits. In *National Bureau of Standards, Applied Math Series*, volume 12, pages 36–38. 1951. Notes by G.E. Forsythe. Reprinted in von Neumann's Collected Works, Vol. 5, Pergamon Press (1963).

A Pseudo-code for the algorithm Q

In this appendix, we give pseudo-code for the efficient execution of the optimal fair bit extractor Q described in the body of the paper. This pseudo-code is broken into two parts. The first part is a pseudo-code function that executes the algorithm Q_1 . In other words, this pseudo-code takes a potentially biased sequence X and outputs the roll of an unbiased die U . The second part is a pseudo-code function that executes the algorithm Q_2 . This pseudo-code takes as input the roll of an unbiased die U , and outputs a unbiased bitstring. Thus, to compute $Q(X)$ for a given sequence X , we simply compute $Q_2(Q_1(X))$.

A.1 Pseudo-code for Q_1

The pseudo-code for Q_1 takes as input a sequence X of real numbers, and outputs a pair (R, S) , representing the roll R obtained from an unbiased die U with S faces. Recall that in order to compute $Q_1(X)$ efficiently, we compute the rolls of a series of small, unbiased dice U_1, U_2, \dots, U_n associated with X , and then combine these small dice into a large, unbiased die U . Recall too that the die U_i has $n - i - 1$ faces, labeled with elements of the rank set $\rho_i(X)$. Since some of these faces may have identical labels, it is convenient in our pseudo-code to represent the roll obtained from U_i as a sub-interval $[l, l + v]$ over the interval $[0, f]$, where $f = n - i + 1$, the number of faces of U_i . The value l may be viewed as a representation of the roll r obtained on U_i . In particular, if the face that comes up on rolling U_i has label r , then l is the number of faces with label values less than r . The value v is equal to the number of faces bearing the label r . Note that v/f is the *a priori* probability of obtaining the roll r on U_i , while l/f is the *a priori* probability of obtaining a roll value less than r .

Example 7 Let us revisit Example 5, in which $X = \{15, 10, 5, 15, 5\}$. For this example, $r_2(X) = \{2, 1, 3, 1\}$, hence the die U_2 has four faces; these bear the labels $\{1, 1, 2, 3\}$. The roll r obtained on U_2 in this example is $r = \text{rank}_2(10) = 2$. Since there is only one face bearing a label less than 2, and since label 2 appears on only one face of U_2 , this roll corresponds to the sub-interval $[2, 3]$ over the interval $[0, 4]$. Note that *a priori* probability of obtaining a roll of 2 on $U_2 = v/f = (3 - 2)/4 = 1/4$.

The algorithm Q_1 maintains a space $[L, F]$ of possible outcomes of the die U . On initialization, this space is equal to $[0, S]$. In each iteration i , this space is narrowed with respect to the roll obtained on die U_i . As explained above, the roll obtained on U_i is represented as a sub-interval $[l, l + v]$ on the interval $[0, f]$. The value of the lower bound l on the roll of U_i serves to update L , the lower bound on the space of possible outcomes of U . In particular, we set $L = l/f * L$ in each iteration. The value v is used to update F , the number of remaining faces that serve as possible outcomes of U . We set $F = v/f * F$ in each iteration. Recall that v/f is the *a priori* probability of obtaining the roll r on die U_i . Thus, as we would expect, the smaller the *a priori* probability associated with r , the more the roll r narrows the possible remaining outcomes of U .

We assume here the existence of a function $\text{factorial}(i)$ which computes $i!$. It is important that this function adhere to the convention that $0! = 1$. Also required is a function $\text{rank}(X[i], X)$ which computes the rank of $X[i]$ in the sequence X .


```

Function  $Q_1(X)$ 
/* Input: a sequence  $X = \{X[1], X[2], \dots, X[n]\}$  of readings from a
    biased die  $D$ 
   Output: a pair  $(R, S)$  representing a roll  $R$  obtained on an unbiased
    die  $U$  with  $S$  sides */

/* Compute rank frequencies */
for  $i = 1$  to  $n$ 
     $fr[i] = 0$ ;
for  $i = 1$  to  $n$ 
     $fr[\text{rank}(X[i], X)]++$ ;

/* Compute number of sides of die  $U$ ; this is  $S = n! / \sum(fr_X(i)!)$  */
 $S = \text{factorial}(n)$ ;
for  $i = 1$  to  $n$ 
     $S = S / \text{factorial}(fr[i])$ ;

 $L = 0$ ;
 $F = S$ ;
/* Main loop */
for  $i = 1$  to  $n - 1$ 
    /* Note that  $U_n$  is a 1-sided die, and can therefore be excluded */

    /* Compute data for die  $U_i$  */
     $l = 0$ ;
    for  $j = 1$  to  $\text{rank}(X[i], X) - 1$ 
         $l = l + fr[j]$ ;
     $v = fr[\text{rank}(X[i], X)]$ ;

    /* Fold data for die  $U_i$  into cumulative roll */
     $f = n - i + 1$ ;
     $L = L + (l/f) * F$ ;
     $F = (v/f) * F$ ;

    /* Update frequency table for remaining measurements */
     $fr[\text{rank}(X[i], X)] --$ ;

 $R = L + 1$ ;
return( $R, S$ );

```

A.2 Pseudo-code for Q_2

Let us now give the pseudo-code for the algorithm Q_2 . This algorithm takes as input a pair (R, S) , where S represents the number of sides of the unbiased die U from which the bits are

being extracted, and $R \in \{1, 2, \dots, S\}$ represents the resulting roll of that die. The algorithm outputs a variable number of bits, or ϕ if no bits are to be returned. We assume a function $\text{binary}(x)$ that returns a conversion of the integer x into a binary representation.

```

Function  $Q_2(R, S)$ 
/* Input:  a pair  $(R, S)$  representing a roll  $R$  obtained on an
           unbiased die  $U$  with  $S$  sides
   Output:  an unbiased bitstring */

 $s_k s_{k-1} \dots s_1 = \text{binary}(S);$ 
 $r_j r_{j-1} \dots r_1 = \text{binary}(R - 1);$ 
 $r_k r_{k-1} \dots r_{j+1} = 00 \dots 0;$ 
for  $i = k$  downto 2
    if  $s_i = 1$  and  $r_i = 0$  then
        return( $r_{i-1} r_{i-2} \dots r_1$ );
return( $\phi$ );

```