Unbiasing Random Bits

Michael Mitzenmacher

1. Introduction

Most computers use a pseudo-random number generator in order to mimic random numbers. While such pseudo-random numbers are sufficient for many applications, they may not do in cases where more secure randomness is needed, such as when you are generating a cryptographic key. For example, years ago the security of the Netscape browser was broken when people found how the seed for the pseudo-random number generator was created. (See "Randomness and the Netscape Browser," by Ian Goldberg and David Wagner, in Dr. Dobb's Journal, January 1996, pp. 66-70.)

When better randomness is required, software can be used to obtain randomness from the computer system, including such behaviors as disk movement, user keystrokes, mouse clicks, or sound recorded by a microphone. While it is clear that many of these physical phenomena can produce random events that are hard to predict, it is not clear how to distill this randomness into something useful, such as random bits that are 0 half the time and 1 half the time. For example, you might try using the number of microseconds between keystrokes to generate random numbers, outputting a 0 if the number of microseconds is even and 1 if the number is odd. While it might be hard to accurately predict the number of microseconds between user keystrokes, it may happen that some users consistently end up with an odd number of microseconds between key strokes 70% of the time.

In this article I will demonstrate a simple means of efficiently extracting random bits from a possibly biased source of bits. I focus on a simple model of a random source: it generates bits that are 0 with probability p and 1 with probability $q = 1 - p$. Here p should be strictly between 0 and 1. Each bit is independent; that is, whether it is 0 or 1 is not correlated with the value of any of the other bits. I want to generate *fair bits* that are independent and are 0 and 1 each with probability ½.

For a more physical interpretation, I suggest the following. You have a coin, and you would like to use it to generate random bits. Unfortunately, the coin may be dented or weighted in some way you do not know about, so it might

be that it comes up heads with some probability p that does not equal ½. Can you use coin to generate fair bits? Interestingly, you can do this even if you do not know the value of p! This procedure has practical applications to software that extracts randomness from biased sources, and also it leads to some fun mathematics. The approach I describe is based on work by Yuval Peres. ("Iterating von Neumann's Procedure for Extracting Random Bits," Annals of Statistics, 20:1, March 1992, pp. 590-597.)

2. Extracting single bits

The first question to consider is how you can use the possibly biased coin to generate just a single fair bit. (For a while, this question proved quite popular at software developer interviews.) For convenience, I will talk about the coin flips as coming up either heads or tails and the bits produced as being 0s and 1s. The key insight you need, which has been attributed to von Neumann, is to use symmetry. Suppose you flip the coin twice. If the coin lands heads and then tails, you should output a 0. This happens with probability pq. If instead the coin lands tails and then heads, you should output a 1. This happens with probability qp = pq. In the case where the coin provides two heads or two tails, you simply start over again. Since the probability you produce a 0 or a 1 is the same for each pair of flips, you must be generating fair bits. Note that our procedure does not even need to know the value of p!

I can write the process described above as a procedure to extract random bits from biased flips. The procedure looks at consecutive pairs of flips and determines if they yield a fair bit. The variable NumFlips represents the number of biased flips available.

```
Function ExtractBits ( Flips[0,NumFlips-1] )
        for (j = 0; j < (NumFlips-1)/2; j++) {
                if ( Flips[2*j] == Heads ) and ( Flips[2*j+1] == Tails ) print 0;
                if ( Flips[2*j] == Tails ) and ( Flips[2*j+1] == Heads ) print 1;
        }
}
```

The above function provides a good first step, but it does not seem very efficient. The function discards pairs of flip when there are two heads (probability p^2) or two tails (probability $q^2 = (1-p)^2$). Using calculus or a graphing calculator reveals that $p^2 + (1 – p)^2$ achieves its minimum value of

½ when p = ½. Hence no matter what the value of p is, the function throws away a pair of flips at least half of the time.

More carefully, suppose I define a function B(p) to represent the average number of fair bits I get per coin flip when the coin lands heads with probability p. You should note that $0 \leq B(p) \leq 1$; I can't get more than 1 fair bit out of even a fair coin! Also, B(p) is meant to represent a long-term average; it ignores issues such as when you have an odd number of coin flips, the last one is useless under this scheme. For every two flips, I get a fair bit with probability 2pq, so B(p) = pq. When p = q = ½, so that my coin is fair and I could conceivably extract 1 full fair bit per flip, B(p) is just ¼.

3. Make more use of symmetry

A better approach continues using symmetry beyond pairs of flips. For example, suppose I flip two heads followed by two tails. In the original extraction scheme, I obtain no fair bits. But if I decide that two heads followed by two tails produces a 0, while two tails followed by two heads produces a 1, then I maintain symmetry while increasing the chances of producing a fair bit.

There is a nice way to visualize how to do this. Consider the original sequence of flips. I build up a new sequence of flips in the following way: whenever I get a pair of flips that are the same in the original sequence of flips, I introduce a new flip of that type into the new sequence. An example is given below:

Original:	H	T	H	H	T	H	T	T	T	T	H	T	T	H	H	H
Bits produced:		0				1						0		1		
New sequence:				H				T		T						H
Bits produced:								0								1

Whenever a pair of flips is heads-tails or tails-heads, I generate a fair bit using von Neumann's approach. Whenver a pair is heads-heads or tails-tails, I add a new coin flip to the new sequence. After I finish with the original sequence of flips, I turn to the new sequence of flips to try to get more fair bits. I append the fair bits from the new sequence to those produced by the original sequence. Here, the final output would be 010101. The new sequence looks for the symmetry between the sequences heads-heads-tails-tails and tails-tails-heads-heads.

There is no reason to stop with just a single new sequence. I can use the new sequence to generate another new sequence, recursively! For example, suppose I change my example above slightly, to the following.

Original:	H	T	H	H	T	H	H	H	T	T	H	T	T	H	T	T
Bits produced:		0				1						0		1		
New sequence:				H				H		T						T
Bits produced:																
New sequence:								H								T
Bits produced:																0

You may notice now that my second level produces no extra fair bits, but if I generate a further new sequence recursively, I obtain one more fair bit.

The recursive variation can be coded as follows:

```
Function ExtractBits ( Flips[0,NumFlips-1] )
        NumNewFlips = 0;
        for (j = 0; j < (n-1)/2; j++) {
                if ( Flips[2*j] == Heads ) and ( Flips[2*j+1] == Tails ) print 0;
                if ( Flips[2*j] == Tails ) and ( Flips[2*j+1] == Heads ) print 1;
                if ( Flips[2*j] == Heads ) and ( Flips[2*j+1] == Heads) {
                        NewFlips[NumNewFlips++] = Heads;
                }
                if ( Flips[2*j] == Tails ) and ( Flips[2*j+1] == Tails) {
                        NewFlips[NumNewFlips++] =Tails;
                }
        }
        if (NumNewFlips ≥ 2) ExtractBits (NewFlips[0,NumNewFlips-1]);
}
```

I can again define a function B(p) to represent the average number of fair bits I get per coin flip when the coin lands heads with probability p. For every two flips, I again get a fair bit with probability 2pq; this adds pq fair bits per coin flip, on average. If I don't get a fair bit, I get a new flip. Recall my coin gave two heads with probability p^2 and two tails with probability q^2, so on average I get $(p^2 + q^2)/2$ additional recursive coin flips per original coin flip. Also, each coin flip at the new level is heads with probability

$p^2/(p^2 + q^2)$ and tails with probability $q^2/(p^2 + q^2)$. (These values sum to 1, and preserve the proper ratio between two heads and two tails.) So now I can write the approriate equation: $B(p) = pq + ((p^2 + q^2)/2)B(p^2/(p^2 + q^2))$. While this equation does not appear to have a simple closed form, you can use it to calculate specific values of B(p) recursively. Moreover, when $p = q = ½$, this equation gives $B(1/2) = ¼ + ¼ B(1/2)$, so $B(1/2) = 1/3$. While this is an improvement for fair coins over the initial scheme, I am still pretty far from the optimal of 1 fair bit per coin flip when the coin is fair. The recursive extraction removes some of the waste, but there is still more to be done.

4. Make even more use of symmetry

There is another symmetry that I can take advantage of that I have not used yet. Consider the cases where the coin lands heads-heads-heads-tails and heads-tails-heads-heads. Both sequences produce one fair bit and one flip for the next sequence in the simple recursive scheme. But I have not taken advantage of the order in which these two events happened; in the first sequence, the fair bit was produced second, and in the second sequence, it was produced first. The symmetry between these two situations can yield another fair bit!

I can give an easy way to extract this extra bit, again using the idea of creating an additional sequence of coin flips. From my original sequence, I can derive a new sequence that gives me a biased coin flip for each consecutive pair of original flips. If the two flips are the same, I get a heads; if the two flips are different, I get a tails. I then apply von Neumann's scheme to the new sequence as well as the original sequence, as in the example below.

Original:	H	T	H	H	T	H	T	T	T	T	H	T	T	H	H	H
Bits produced:		0				1						0		1		
New sequence:		H		T		H		T		T		H		H		T
Bits produced:				0				0				1				0

I can again glue together the two outputs to obtain 01010010.

Now I can also use the additional sequence from the last section. So when I start with an original sequence, I use it to derive two further sequences, as

shown below. It turns out that if I glue all the bits together, I get independent and fair bits.

Original:	H	T	H	H	T	H	T	T	T	T	H	T	T	H	H	H
Bits produced:		0				1						0		1		
New sequence A:		H		T		H		T		T		H		H		T
Bits produced:				0				0				1				0
New sequence B:				H				T		T						H
Bits produced:								0								1

Of course, I can go further by recursively extracting more bits from each sequence. That is, each new sequence should generate two further new sequences, and so on. This recursive construction is easy to code.

```
Function ExtractBits ( Flips[0,NumFlips-1] )
        NumNewFlipsA = 0;
        NumNewFlipsB = 0;
        for (j = 0; j < (NumFlips-1)/2; j++) {
                if ( Flips[2*j] == Heads ) and ( Flips[2*j+1] == Tails ) {
                        print 0;
                        NewFlipsA[NumNewFlipsA++] = Heads;
                }
                if ( Flips[2*j] == Tails) and ( Flips[2*j+1] == Heads ) {
                        print 1;
                        NewFlipsA[NumNewFlipsA++] = Heads;
                }
                if ( Flips[2*j] == Heads ) and ( Flips[2*j+1] == Heads) {
                        NewFlipsB[NumNewFlipsB++] = Heads;
                        NewFlipsA[NumNewFlipsA++] = Tails;
                }
                if ( Flips[2*j] == Tails ) and ( Flips[2*j+1] == Tails) {
                        NewFlipsB[NumNewFlipsB++] = Tails;
                        NewFlipsA[NumNewFlipsA++] = Tails;
                }
        }
        if (NumNewFlipsA ≥ 2) ExtractBits (NewFlipsA[0,NumNewFlipsA-
1]);
        if (NumNewFlipsB ≥ 2) ExtractBits (NewFlipsB[0,NumNewFlipsA-
1]);
}
```

In all of the procedures I have given so far, I have designed them recursively, so the bits from one sequence are output before the bits from the derived sequences are output. You may be wondering if this is important. In fact you must be somewhat careful to make sure that you do not introduce correlations by ordering the bits in an unusual fashion. The recursive approach I have described is known to be safe, so I recommend sticking with it.

I can again write an equation that determines the long-term average number of bits produced per flip. The equation is similar to the previous case, except now for every two flips we also get an additional flip in one of our derived sequences. This flip is heads with probability $p^2 + q^2$, since it comes up heads whenever the pair of flips are the same. Hence I have the resulting equation:

$$B(p) = pq + ((p^2 + q^2)/2)B(p^2/(p^2 + q^2)) + (1/2)B(p^2 + q^2).$$

If I again test B(1/2), I find B(1/2) = ¼ + ¼B(1/2) + ½B(1/2), so B(1/2) = 1. Now if you flip a fair coin, you expect in the long run to get out 1 fair bit per flip using this recursive procedure. We are finally doing essentially as good as we could hope for!

In fact in the limit this process extracts the maximum number of fair bits possible for every value of p. To prove this requires knowing some information theory. You need to know that the maximum rate at which fair bits can be produced from bias bits is given by the entropy function, usually denoted by $H(p) = p\log_2(1/p) + q\log_2(1/q)$. The complex looking equation above has a very straightforward solution; B(p) equals the entropy H(p). You can verify this by plugging B(p) = H(p) into the recurrence. It is easiest to first calculate term by term, and to use the fact that when p + q = 1, you have $1 - p^2 + q^2 = 2pq$. Suppose B(p) = H(p). Then we first calculate easier expressions for the terms on the right hand side.

$$\left(p^2+q^2/2\right)B\left(p^2/p^2+q^2\right)$$

$$=\left(p^2+q^2/2\right)\left[\left(p^2/p^2+q^2\right)\log_2\left(p^2+q^2/p^2\right)+\left(q^2/p^2+q^2\right)\log_2\left(p^2+q^2/q^2\right)\right]$$

$$=\left(p^2/2\right)\left(\log_2\left(p^2+q^2\right)-\log_2\left(p^2\right)\right)+\left(q^2/2\right)\left(\log_2\left(p^2+q^2\right)-\log_2\left(q^2\right)\right)$$

$$=\left(p^2+q^2/2\right)\log_2\left(p^2+q^2\right)-p^2\log_2 p-q^2\log_2 q$$

$$1/2\,B(p^2+q^2)$$

$$=\left(p^2+q^2/2\right)\log_2\left(1/p^2+q^2\right)+\left(1-p^2+q^2/2\right)\log_2\left(1/1-p^2+q^2\right)$$

$$=-\left(p^2+q^2/2\right)\log_2\left(p^2+q^2\right)-(pq)\log_2(2pq)$$

$$=-\left(p^2+q^2/2\right)\log_2\left(p^2+q^2\right)-pq-(pq)\log_2 p-(pq)\log_2 q$$

Now we check that the right hand side comes out to H(p)=B(p).

$$pq+\left(p^2+q^2/2\right)B\left(p^2/p^2+q^2\right)+1/2\,B(p^2+q^2)$$

$$=-p^2\log_2 p-q^2\log_2 q-(pq)\log_2 p-(pq)\log_2 q$$

$$=-p(p+q)\log_2(p)-q(p+q)\log_2(q)$$

$$=p\log_2(1/p)+q\log_2(1/q)=H(p)$$

5. Conclusions

Although I have given some basic psuedo-code, you might find it interesting to try to write more efficient versions of the code on your own. There is a collections of implementations and related links at http://www.ciphergoth.org/software/unbiasing, and Peres's work has been the subject of discussion on the sci.crypt newsgroup.