CS 124 Lecture 1

January 27, 2020

Coin Flipping

Coin: heads w/ probability p. How to make it fair?

Symmetry Algorithm

$$\left.\begin{array}{l} HT: \text{heads} \\ TH: \text{tails} \end{array}\right\} \text{same probability, } pq$$

HH: ignore

TT : ignore

Algorithm performance (time to a fair flip):

For two flips: $t = 2pq \cdot 2 + (1-2pq)(2+t)$

$$= \cancel{4pq} + 2 - \cancel{4pq} + (1-2pq)t$$

$$t = 1/pq$$

Geometric random variable: $\frac{1}{2pq} \cdot 2 = \frac{1}{pq}$

Better Algorithm

$p = 2/3$ 8/9 success probability

$HH \,] \to$ Heads

$\left.\begin{array}{l} HT \\ TH \end{array}\right] \to$ Tails

$TT \to$ Repeat

After 2^k flips, $(1-p^{2^k}-q^{2^k})$ probability we still need to flip

$p = ?$

$HH : p^2$	HHTT : heads	HHHHTTTT : heads
$HT : p(1-p)$	heads	
$TH : p(1-p)$	tails	
$TT : (1-p)^2$	TTHH : tails	TTTT HHHH : tails

Syllabus

Midterm: Monday before spring break

Back to Coin Flip

prob of H is p^2

generated w/ prob $2pq$ every 2 flips

```
        H                  H          H
HH  HT  TT  HH  HT  HH  HT        Level 0
H       T   H       H             Level 1a
H   T   H   H   T   H   T         Level 1b
```

symmetry

heads if discarded
tails if not

probability of H is p^2+q^2

A(p) = average # of unbiased coin flips generated per biased coin flip

$$= pq + A\left(\frac{p^2}{p^2+q^2}\right) \times \frac{p^2+q^2}{2} + A(p^2+q^2) \times \frac{1}{2}$$

most amount of information you can get

CHECK:

$$A(1/2) = \frac{1}{2} \times \frac{1}{2} + A(1/2) \times \frac{1}{4} + A(1/2) \times \frac{1}{2} \rightarrow \frac{A(1/2)}{4} = \frac{1}{4}$$

$A(1/2) = 1$ ✓

Entropy

Binary crossentropy function:

$$H(p) = p \log_2(p) - (1-p) \log_2(1-p)$$

$0 \leq p \leq 1$

$H(1/2) = 1$

Theoretically, $A(p) = H(p)$

Tossing a Biased Coin

Michael Mitzenmacher[*]

When we talk about a coin toss, we think of it as unbiased: with probability one-half it comes up heads, and with probability one-half it comes up tails. An ideal unbiased coin might not correctly model a real coin, which could be biased slightly one way or another. After all, real life is rarely fair.

This possibility leads us to an interesting mathematical and computational question. Is there some way we can use a biased coin to efficiently simulate an unbiased coin? Specifically, let us start with the following problem:

Problem 1. Given a biased coin that comes up heads with some probability greater than one-half and less than one, can we use it to simulate an unbiased coin toss?

A simple solution, attributed to von Neumann, makes use of symmetry. Let us flip the coin twice. If it comes up heads first and tails second, then we call it a 0. If it comes up tails first and heads second, then we call it a 1. If the two flips are the same, we flip twice again, and repeat the process until we have a unbiased toss. If we define a round to be a pair of flips, it is clear that we the probability of generating a 0 or a 1 is the same each round, so we correctly simulate an unbiased coin. For convenience, we will call the 0 or 1 produced by our simulated unbiased coin a *bit*, which is the appropriate term for a computer scientist.

Interestingly enough, this solution works regardless of the probability that the coin lands heads up, even if this probability is unknown! This property seems highly advantageous, as we may not know the bias of a coin ahead of time.

Now that we have a simulation, let us determine how efficient it is.

Problem 2. Let the probability that the coin lands heads up be p and the probability that the coin lands tails up be $q = 1 - p$. On average, how many flips does it take to generate a bit using von Neumann's method?

Let us develop a general formula for this problem. If each round takes exactly f flips, and the probability of generating a bit each round is e, then the expected number of total flips t satisfies a simple equation. If we succeed in the first round, we use exactly f flips. If we do not, then we have flipped the coin f times, and because it is as though we have to start over from the beginning again, *the expected remaining number of flips is still t*. Hence t satisfies

$$t = ef + (1-e)(f+t).$$

or, after simplifying

$$t = f/e.$$

Using von Neumann's strategy, each round requires two flips. Both a 0 and a 1 are each generated with probability pq, so a round successfully generates a bit with probability $2pq$. Hence the average number of flips required to generate a bit is $f/e = 2/2pq = 1/pq$. For example, when $p = 2/3$, we require on average $9/2$ flips.

We now know how efficient von Neumann's initial solution is. But perhaps there are more efficient solutions? First let us consider the problem for a specific probability p.

[*]Digital Equipment Corporation, Systems Research Center, Palo Alto, CA.

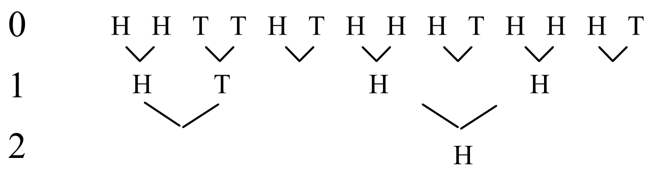

Figure 1: The Multi-Level Strategy.

Problem 3. Suppose we know that we have a biased coin that comes up heads with probability $p = 2/3$. Can we generate a bit more efficiently than by von Neumann's method?

We can do better when $p = 2/3$ by matching up the possible outcomes a bit more carefully. Again, let us flip the coin twice each round, but now we call it a 0 if two heads come up, while we call it a 1 if the tosses come up different. Then we generate a 0 and a 1 each with probability $4/9$ each round, instead of the $2/9$ using von Neumann's method. Plugging into our formula for $t = f/e$, we use $f = 2$ flips per round and the probability e of finishing each round is $8/9$. Hence the average number of coin flips before generating a bit drops to $9/4$.

Of course, we made strong use of the fact that p was 2/3 to obtain this solution. But now that we know that more efficient solutions might be possible, we can look for methods that work for any p. It would be particularly nice to have a solution that, like von Neumann's method, does not require us to know p in advance.

Problem 4. Improve the efficiency of generating a bit by considering the first four biased flips (instead of just the first two).

Consider a sequence of four flips. If the first pair of flips are H T or T H, or the first pair of flips are the same but the second pair are H T or T H, then we use von Neumann's method. We can improve things, however, by pairing up the sequences H H T T and T T H H; if the first sequence appears we call it a 0, and if the second sequence appears we call it a 1. That is, if both pairs of flips are the same, but the pairs are different, then we can again decide using von Neumann's method, *except that we consider the order of the pairs of flips*. (Note that our formula for the average number of flips no longer applies, since we might end in the middle of our round of four flips.)

Once we have this idea, it seems natural to extend it further. A picture here helps– see Figure 1. Let us call each flip of the actual coin a Level 0 flip. If Level 0 flips $2j-1$ and $2j$ are different, then we can use the order (heads-tails or tails-head) to obtain a bit. (This is just von Neumann's method again.) If the two flips are the same, however, then we will think of them as providing us with what we shall call a Level 1 flip. If Level 1 flips $2j-1$ and $2j$ are different, again this gives us a bit. But if not, we can use it to get a Level 2 flip, and so on. We will call this the Multi-Level strategy.

Problem 5a. What is the probability we have not obtained a bit after flipping a biased coin 2^k times using the Multi-Level strategy?

Problem 5b (HARD!). What is the probability we have not obtained a bit after flipping a biased coin ℓ times using the Multi-Level strategy??

Problem 5c (HARDEST!)**.** How many biased flips does one need on average before obtaining a bit using the Multi-Level strategy?

For the first question, note that the only way the Multi-Level strategy will not produce a bit after 2^k tosses is if all the flips have been the same. This happens with probability $p^{2^k} + q^{2^k}$.

Using this, let us now determine the probability the Multi-Level strategy fails to produce a bit in the first ℓ bits, where ℓ is even. (The process never ends on an odd flip!) Suppose that $\ell = 2^{k_1} + 2^{k_2} + \dots 2^{k_m}$, where $k_1 > k_2 > \dots > k_m$. First, the Multi-Level strategy must last the first 2^{k_1} flips, and we have already determined the probability that this happens. Next, the process must last the next 2^{k_2} flips. For this to happen, all of the next 2^{k_2} flips have to be the same, *but they do not have to be the same as the first* 2^{k_1} *flips.* Similarly, each of the next 2^{k_3} flips have to be the same, and so on. Hence the probability of not generating a bit in ℓ flips is

$$\prod_{i=1}^{m} (p^{2^{k_i}} + q^{2^{k_i}}).$$

Given the probability that the Multi-Level strategy requires at least ℓ flips, calculating the average number of flips t_2 before the Multi-Level strategy produces a bit still requires some work. Let $P(\ell)$ be the probability that the Multi-Level strategy takes exactly ℓ flips to produce a bit, and let $Q(\ell)$ be the probability that the Multi-Level strategy takes more than ℓ flips. Of course, $P(\ell) = 0$ unless ℓ is even, since we cannot end with an odd number of flips! Also, for l even it is clear than $P(\ell) = Q(\ell - 2) - Q(\ell)$, since the right hand side is just the probability that the Multi-Level strategy takes ℓ flips. Finally, we previously found that $Q(\ell) = \prod_{i=1}^{m} (p^{2^{k_i}} + q^{2^{k_i}})$ above.

The average number of flips is, by definition,

$$t_2 = \sum_{\ell \geq 2, \ell \text{ even}} P(\ell) \cdot \ell$$

We change this into a formula with the values $Q(\ell)$, since we already know how to calculate them.

$$\begin{aligned} t_2 &= \sum_{\ell \geq 2, \ell \text{ even}} P(\ell) \cdot \ell \\ &= \sum_{\ell \geq 2, \ell \text{ even}} (Q(\ell - 2) - Q(\ell)) \cdot \ell \end{aligned}$$

Now we use a standard "telescoping sum" trick; we re-write the sum by looking at the coefficient of each $Q(\ell)$.

$$\begin{aligned} t_2 &= \sum_{\ell \geq 2, \ell \text{ even}} P(\ell) \cdot \ell \\ &= \sum_{\ell \geq 2, \ell \text{ even}} (Q(\ell - 2) - Q(\ell)) \cdot \ell \\ &= \sum_{\ell \geq 0, \ell \text{ even}} Q(\ell)(\ell + 2 - \ell) \\ &= 2 \sum_{\ell \geq 0, \ell \text{ even}} Q(\ell) \end{aligned}$$

This gives an expression for the average number of biased flips we need to generate a bit. It turns out this sum can be simplified somewhat, as using the expression for $Q(\ell)$ above we have

$$2 \sum_{\ell \geq 0, \ell \text{ even}} Q(\ell) = 2 \prod_{k \geq 1} (1 + p^{2^k} + q^{2^k}).$$

Up to this point, we have tried to obtain just a single bit using our biased coin. Instead, we may want to obtain several bits. For example, a computer scientist might need a collection of bits to apply a randomized

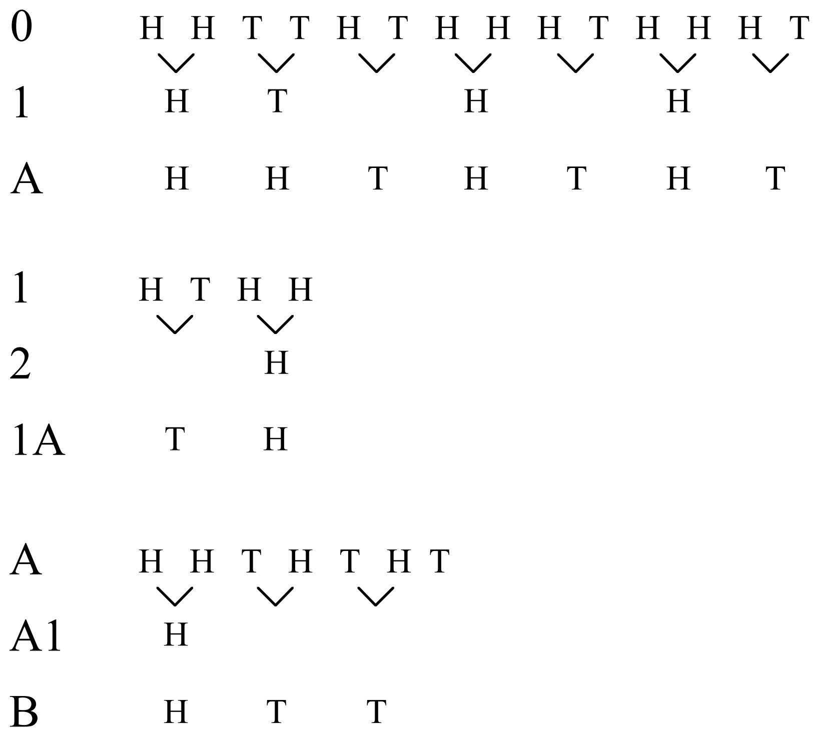

Figure 2: The Advanced Multi-Level Strategy. Each sequence generates two further sequences. Bits are generated by applying von Neumann's rule to the sequences in some fixed order.

algorithm, but the only source of randomness available might be a biased coin. We can obtain a sequence of bits with the Multi-Level strategy in the following way: we flip the biased coin a large number of times. Then we run through each of the levels, producing a bit for each heads-tails or tails-heads pair. This works, but there is still more we can do if we are careful.

Problem 6. Improve upon the Multi-Level strategy for obtaining bits from a string of biased coin flips. Hint: consider recording whether each pair of flips provides a bit via von Neumann's method or not.

The Multi-Level strategy does not take advantage of *when* each level provides us with a bit. For example, in the Multi-Level strategy, the sequences H H H T and H T H H produce the same single bit. However, since these two sequences occur with the same probability, we can pair up these two sequences to provide us with a second bit; if the first sequence comes up, we consider that a 0, and if the second comes up, we can consider it a 1.

To extract this extra randomness, we expand the Multi-Level strategy to the Advanced Multi-Level strategy. Recall that in the Multi-Level strategy, we used Level 0 flips to generate a sequence of Level 1 flips. In the Advanced Multi-Level strategy, we determine two sequences from Level 0. The first sequence we extract will be Level 1 from the Multi-Level Strategy. For the second sequence, which we will call Level A, flip j records whether flips $2j-1$ and $2j$ are the same or different in Level 0. If the flips are different, then the flip in Level A will be tails, and otherwise it will be heads. (See Figure 2.) Of course, we can repeat this process, so from each of both Level 1 and Level A, we can ge two new sequences, and so on. To extract a sequence of bits, we go through all these sequences in a fixed order and use von Neumann's method.

How good is the Advanced Mult-Level Strategy? It turns out that it is essentially as good as you can possibly get. This is somewhat difficult to prove, but we can provide a rough sketch of the argument.

Let $A(p)$ be the average number of bits produced for each biased flip, when the coin comes us heads with probability p. For convenience, we think of ths average over an infinite number of flips, so that we don't have to worry about things like the fact that if we end on an odd flip, it cannot help us. We first determine an equation that describes $A(p)$.

Consider a consecutive pair of flips. First, with probability $2pq$ we get H T or T H, and hence get out one bit. So on average von Neumann's trick alone yields pq bits per biased flip. Second, for every two flips, we always get a single corresponding flip for Level A. Recall that we call a flip on Level A heads if the two flips on Level 0 are the same and tails if the two flips are different. Hence for Level A, a flip is heads with probability p^2+q^2. This means that for every two flips on Level 0, we get one flip on Level A, with a coin that has a different bias– it is heads with probability p^2+q^2. So for every two biased Level 0 flips, we get (on average) $A(p^2+q^2)$ bits from Level A. Finally, we get a flip for Level 1 whenever the two flips are the same. This happens with probability p^2+q^2. In this case, the flip at the next level is heads with probability $p^2/(p^2+q^2)$. So on average each two Level 0 flips yields (p^2+q^2) Level 1 flips, where the Level 1 flips again have a different bias, and thus yield $A(p^2/(p^2+q^2))$ bits on average. Putting this all together yields:

$$A(p) = pq + \frac{1}{2}A(p^2+q^2) + \frac{1}{2}(p^2+q^2)A\left(\frac{p^2}{p^2+q^2}\right).$$

Problem 7. What is $A(1/2)$?

Plugging in yields $A(1/2) = 1/4 + A(1/2)/2 + A(1/2)/4$, and hence $A(1/2) = 1$. Note that $A(1/2)$ is the average number of bits we obtain per flip when we flip a coin that comes up heads with probability $1/2$. This result is somewhat surprising: it says the Advanced Multi-Level strategy extracts (on average, as the number of flips goes to infinity) 1 bit per flip of an unbiased coin, and this is clearly the best possible! This gives some evidence that the Advanced Multi-Level strategy is doing as well as can be done.

You may wish to think about it to convince yourself there is no other randomness lying around that we are not taking advantage of. Proving that the Advanced Multi-Level strategy is optimal is, as we have said, rather difficult. (See the paper "Iterating von Neumann's Procedure" by Yuval Peres, in *The Annals of Statistics*, 1992, pp. 590-597.) It helps to know the average rate that we could ever hope to extract bits using a biased coin that comes up heads with probability p and tails with probability $q = 1-p$. It turns out the correct answer is given by the entropy function $H(p) = -p\log_2 p - q\log_2 q$. (Note $H(1/2) = 1$; see Figure 3.) We will not even attempt to explain this here; most advanced probability books explain entropy and the entropy function. Given the entropy function, however, we may check that our recurrence for $A(p)$ is satisfied by $A(p) = H(p)$.

Problem 8. Verify that $A(p) = H(p)$ satisfies the recurrence above.

This derivation itself is non-trivial! Let us plug in $A(p) = H(p)$ on the right hand side of the recurrence and simplify. First,

$$\begin{aligned}
\frac{1}{2}H(p^2+q^2) &= -\frac{1}{2}(p^2+q^2)\log_2(p^2+q^2) - \frac{1}{2}(1-p^2-q^2)\log_2(1-p^2-q^2) \\
&= -\frac{1}{2}(p^2+q^2)\log_2(p^2+q^2) - pq\log_2(2pq) \\
&= -\frac{1}{2}(p^2+q^2)\log_2(p^2+q^2) - pq - pq\log_2 p - pq\log_2 q,
\end{aligned}$$

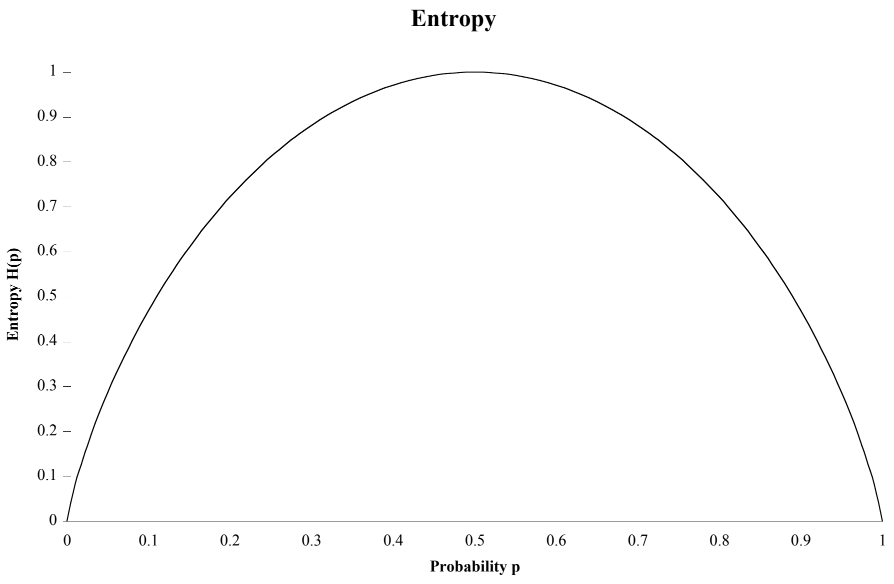

Figure 3: A Graph of the Entropy $H(p)$ vs p.

where we have used the fact that $1-p^2-q^2=2pq$. Second,

$$\begin{aligned}
\frac{1}{2}(p^2+q^2)H(\frac{p^2}{p^2+q^2}) &= -\frac{1}{2}p^2\log_2\frac{p^2}{p^2+q^2}-\frac{1}{2}q^2\log_2\frac{q^2}{p^2+q^2} \\
&= -\frac{1}{2}p^2\log_2 p^2+\frac{1}{2}p^2\log_2(p^2+q^2)-\frac{1}{2}q^2\log_2 q^2+\frac{1}{2}q^2\log_2(p^2+q^2) \\
&= -p^2\log_2 p-q^2\log_2 q+\frac{1}{2}(p^2+q^2)\log_2(p^2+q^2)
\end{aligned}$$

Now the right hand side simplifies dramatically:

$$\begin{aligned}
pq+\frac{1}{2}A(p^2+q^2)+\frac{1}{2}(p^2+q^2)A(\frac{p^2}{p^2+q^2}) &= pq-\frac{1}{2}(p^2+q^2)\log_2(p^2+q^2)-pq-pq\log_2 p-pq\log_2 q \\
&\quad -p^2\log_2 p-q^2\log_2 q+\frac{1}{2}(p^2+q^2)\log_2 p^2+q^2 \\
&= -pq\log_2 p-pq\log_2 q-p^2\log_2 p-q^2\log_2 q \\
&= -p(p+q)\log_2 p-q(p+q)\log_2 q \\
&= -p\log_2 p-q\log_2 q \\
&= H(p)
\end{aligned}$$

Notice that we used the fact that $p+q=1$ in the third line from the bottom. As the right hand side simplifies to $H(p)$, the function $H(p)$ satisfies the recurrence for $A(p)$.

We hope this introduction to biased coins leads you to more questions about randomness and how to use it. Now, how do you simulate an unbiased die with a biased die...

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

$\frac{1}{2}$ when $p = \frac{1}{2}$. 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 $p = q = \frac{1}{2}$, so that my coin is fair and I could conceivably extract 1 full fair bit per flip, $B(p)$ is just $\frac{1}{4}$.

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 $\geq$ 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.

$$\left(\frac{p^2+q^2}{2}\right)B\left(\frac{p^2}{p^2+q^2}\right)$$

$$=\left(\frac{p^2+q^2}{2}\right)\left[\left(\frac{p^2}{p^2+q^2}\right)\log_2\left(\frac{p^2+q^2}{p^2}\right)+\left(\frac{q^2}{p^2+q^2}\right)\log_2\left(\frac{p^2+q^2}{q^2}\right)\right]$$

$$=\left(\frac{p^2}{2}\right)\left(\log_2\left(p^2+q^2\right)-\log_2\left(p^2\right)\right)+\left(\frac{q^2}{2}\right)\left(\log_2\left(p^2+q^2\right)-\log_2\left(q^2\right)\right)$$

$$=\left(\frac{p^2+q^2}{2}\right)\log_2\left(p^2+q^2\right)-p^2\log_2 p-q^2\log_2 q$$

$$\tfrac{1}{2}B(p^2+q^2)$$

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

$$=-\left(\frac{p^2+q^2}{2}\right)\log_2\left(p^2+q^2\right)-(pq)\log_2(2pq)$$

$$=-\left(\frac{p^2+q^2}{2}\right)\log_2\left(p^2+q^2\right)-pq-(pq)\log_2 p-(pq)\log_2 q$$

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

$$pq+\left(\frac{p^2+q^2}{2}\right)B\left(\frac{p^2}{p^2+q^2}\right)+\tfrac{1}{2}B(p^2+q^2)$$

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

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

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

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.