Solutions to Homework 2

Spring 2008

1. Write the entropy as $-\sum_{i=1}^{m} p_i \log_2 p_i$, remembering (Figure 1) that $0 \log 0 = 0$.

   (a) $0 \leq p_i \leq 1$ for all i, so $\log_2 p_i \leq 0$, and $-p_i \log_2 p_i \geq 0$. Hence, $H \geq 0$, being a sum of non-negative terms.

   (b) There are several ways to do this one. Using the hint, write

   $$p_i = \frac{1}{m} + \delta_i \tag{1}$$

   Since $\sum_i p_i = 1$, we must also have $\sum_i \delta_i = 0$. What we'd like to show is that H is maximized when all the $\delta_i = 0$. At this point, we have a situation like that in note about maximizing likelihood for Markov chain. We can either solve for one of the δ_i in terms of the others, and maximize with respect to them, or we can use Lagrange multipliers. Let's do that: the Lagrangian is

   $$\mathcal{L} = -\sum_{i=1}^{m} \left(\frac{1}{m} + \delta_i\right) \log_2 \left(\frac{1}{m} + \delta_i\right) + \lambda \sum_i \delta_i \tag{2}$$

   Taking the derivative of $\mathcal{L}$ with respect to δ_i,

   $$\begin{aligned} 0 &= -\left[\log_2 \frac{1}{m} + \delta_i + \frac{1}{\ln 2} \frac{\frac{1}{m} + \delta_i}{\frac{1}{m} + \delta_i}\right] + \lambda & (3) \\ &= -\log_2 \left(\frac{1}{m} + \delta_i\right) - \frac{1}{\ln 2} + \lambda & (4) \\ \log_2 \left(\frac{1}{m} + \delta_i\right) &= -\frac{1}{\ln 2} + \lambda & (5) \\ \frac{1}{m} + \delta_i &= 2^{-\frac{1}{\ln 2} + \lambda} & (6) \\ \delta_i &= -\frac{1}{m} + 2^{-\frac{1}{\ln 2} + \lambda} & (7) \end{aligned}$$

   On the other hand, taking the derivative of $\mathcal{L}$ with respect to λ, we recover the constraint:

   $$\sum_{i=1}^{n} \delta_i = 0 \tag{8}$$

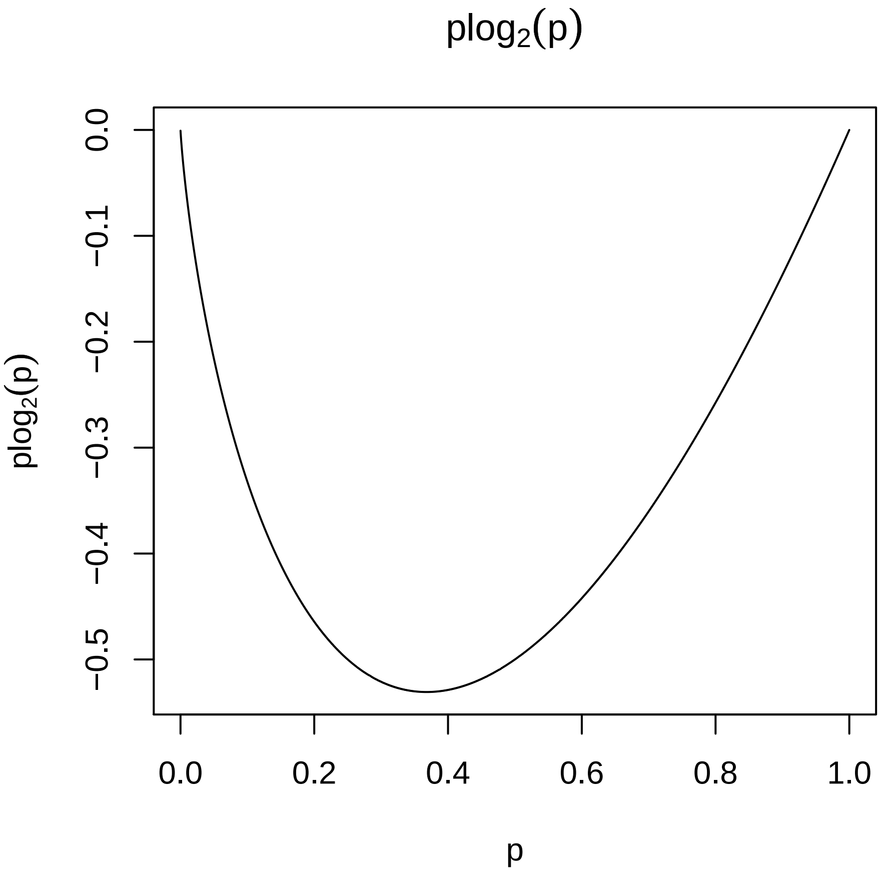

Figure 1: Plot of $p \log_2 p$ versus p. Note that $0 \log 0 = 0$.

Plugging in Eq. 7, and noticing that it is the *same* for all i,

$$0 = \sum_{i=1}^{m} -\frac{1}{m} + 2^{-\frac{1}{\ln 2}+\lambda} \tag{9}$$

$$= m\left[-\frac{1}{m} + 2^{-\frac{1}{\ln 2}+\lambda}\right] \tag{10}$$

$$= -\frac{1}{m} + 2^{-\frac{1}{\ln 2}+\lambda} \tag{11}$$

so (going back to Eq. 7 again) $\delta_i = 0$, as desired.

2. We need to do three things here. First, we need to generate symbol sequences from the logistic map. Second, we need to count the number of times each word of length L appears, and how often it is immediately followed by each other word of length L. Finally, we need to somehow use those counts to test independence. (Notice, by the way, that the second part, about gathering count statistics of words, is something we're going to have to use a lot in the other problems.) I'll do this by writing a bunch of small functions, each of which solves a small piece of the problem; this makes it easier to test that they're doing what they are supposed to, and to re-use them later, if need be.

   The first part uses some code for the logistic map (taken from `03.R`, plus the discretization function.

```
logistic.map <- function(x,r) {
  return(4*r*x*(1-x))
}

logistic.map.ts <- function(timelength,r,initial.cond=NULL) {
  x <-vector(mode="numeric",length=timelength)
  if(is.null(initial.cond)) {
    x[1] <-runif(1)
  } else {
    x[1] <-initial.cond
  }
  for (t in 2:timelength) {
    x[t] = logistic.map(x[t-1],r)
  }
  return(x)
}

logistic.genpart <- function(x) {
  # Apply the generating partition of the logistic map to a vector of
  # real values
  # So return "L" at each position where x < 0.5 and "R" elsewhere
  ifelse(x<0.5,'L','R')
}

logistic.symbseq <- function(timelength,r) {
  # Get a real-valued trajectoy from the logistic map
  x <- logistic.map.ts(timelength,r)
  # Apply the generating partition and return that
  s <- logistic.genpart(x)
  return(s)
}
```

The second part needs us to take a window of length L (not to be confused with the symbol "L"!), slide it along the symbol sequence, and record all the patterns we see. We also need to keep track of which pattern followed that one, in the next L block. Start with a function to find all the different blocks, in order.[1]

```
symbseq.to.blocks <- function(s,L,mode="vectorized") {
  # Take a symbol sequence and return a list of the (overlapping) blocks
  # of length L it contains
  n <- length(s)
  # A length L block can't start at any position whose index is greater
  # than n-L+1 (though it could start there).
  # Should really check that this is < n, and return an error if not!
  max.index <- n -L+1
  # We'll need to repeatedly take a block from the sequence and collapse it
  # down into a string, so make that into a function
  collapser <- function(i) {paste(s[i:(i+L-1)],collapse="")}
  # Now: chop the sequence into overlapping blocks, collapse each one down,
  # all the strings in a vector
  # Natural way to do this is by iteration, but R is much happier (several
  # times faster) with the "vectorized" alternative
  if (mode == "vectorized") {
    blocks <- sapply(1:max.index,collapser)
  } else {
    blocks <- NULL # Null list
    for (i in 1:max.index) {
      blocks <- c(blocks, collapser(i))
    }
  }
  return(blocks)
}
```

Now we need to get not just each block of length L, but also the following block of length L. We'll do this by taking the complete list of L-blocks and splitting it into two parts (which in general will overlap).

[1]The code below differs slightly from the one in the handout with the partial solutions, because I realized explicit iteration was slowing things down, and it could be vectorized.

```
symbseq.to.successive.blocks <- function(s,L) {
  n <- length(s)
  # Produce the complete list of length-L blocks
  all.blocks <- symbseq.to.blocks(s,L)
  # The "leaders" begin at positions 1, 2, ... n-2L+1 (because there
  # needs to be another, following block of length L after each of them)
  max.index.leaders <- n-2*L+1
  # The "followers" begin at positions L+1, L+2, ... n-L+1 (because there
  # needs to be a "leader" block of lenght L before each of them)
  min.index.followers <- L+1
  max.index.followers <- n-L+1
  leaders <- all.blocks[1:max.index.leaders]
  followers <- all.blocks[min.index.followers:max.index.followers]
  return(list(leaders=leaders,followers=followers))
}
```

At this point it's a good idea to check that everything is working right with a small example.

```
> ss <- c("L","R","L","R","L","R","L")
> symbseq.to.blocks(ss,2)
[1] "LR" "RL" "LR" "RL" "LR" "RL"
> symbseq.to.successive.blocks(ss,2)
$leaders
[1] "LR" "RL" "LR" "RL"

$followers
[1] "LR" "RL" "LR" "RL"
```

You can check by hand that the code works on this example, and on this one:

```
> rr <- c("L","L","L","R","L","R","R")
> symbseq.to.blocks(rr,2)
[1] "LL" "LL" "LR" "RL" "LR" "RR"
> symbseq.to.successive.blocks(rr,2)
$leaders
[1] "LL" "LL" "LR" "RL"

$followers
[1] "LR" "RL" "LR" "RR"
```

Ideally at this point I'd check an $L = 3$ case, but I'm just the teacher here.

Finally, we need to test whether the follower blocks are statistically independent of the leader blocks. The standard way to test whether two discrete random variables are independent is to use the χ^2 ("chi-squared") test. If you need a refresher on the theory of this test, I'd suggest either Wikipedia, or (better yet) Larry Wasserman's *All of Statistics*. Fortunately, this is built in to `R`, in the imaginatively-named function `chisq.test`. It needs to be given a contingency table, but there is a function to build that, called `table`. Here's how `table` works:

```
> table(symbseq.to.successive.blocks(ss,2))
       followers
leaders LR RL
     LR  2  0
     RL  0  2
```

and here's how the testing function works:

```
> chisq.test(table(symbseq.to.successive.blocks(ss,2)))

  Pearson's Chi-squared test with Yates' continuity correction

data:  table(symbseq.to.successive.blocks(ss, 2))
X-squared = 1, df = 1, p-value = 0.3173

Warning message:
In chisq.test(table(symbseq.to.successive.blocks(ss, 2))) :
  Chi-squared approximation may be incorrect
```

`chisq.test` is giving us a warning here, because the χ^2 approximation to the distribution of the test statistic is only valid if there are a fairly large number of counts for each cell in the table. The usual rule of thumb is that the expected number of counts must be at least 5; let's say 10 to be safe. Each cell in the table corresponds to a word of length $2L$, and we expect (for IID coin-tossing) that each such word is equally likely, so we want $10 = n/2^{2L}$, or $n = 10 \times 2^{2L}$.

Putting everything together, then, we can write the following program.

```
logistic.map.independence.test <- function(L,n=min(1e4,10*(2^(2*L))),r=1) {
  s <- logistic.symbseq(n,r)
  successive.blocks <- symbseq.to.successive.blocks(s,L)
  my.tab <- table(successive.blocks)
  my.test <- chisq.test(my.tab)
  return(list(p.value=my.test$p.value,test=my.test,count.table=my.tab))
}
```

The default value for n is set so that the program won't take forever to run if you should accidentally input a large L — but that's only a default so it can be over-ridden. Returning the full test results and the count table as well as the p-value is not strictly necessary but doesn't hurt. Working for different values of r is also a bonus (but just as easy as not including it).

How do we know if this is working? If the "leader" and "follower" blocks *are* independent, then the p value of the test should be uniformly distributed on $[0, 1]$, and their CDF should be a straight diagonal line. Let's check that by re-running the test a bunch of times and plotting the empirical CDF (Figure 2).

```
> plot(ecdf(replicate(1000,logistic.map.independence.test(2)$p.value)),
  xlab="Nominal p-value",ylab="True p-value",main="Distribution of
  p-values")
> abline(a=0,b=1,col="blue",lty=2)
```

3. Recall that the topological entropy rate is defined as

$$h_0 \equiv \lim_{L\to\infty} \frac{1}{L} \log W_L \tag{12}$$

where W_L is the number of allowed words of length L.

(a) There are at least three ways to do this. The simplest one is to just count the number of distinct *observed* words of length L, $\widehat{W}_L$, and estimate by division:

$$\widehat{h}_0^{division} \equiv \frac{1}{L} \log \widehat{W}_L \tag{13}$$

for some large L, as our estimate of h_0.

The second way is to notice that if the limit exists, then for large L we must have

$$\log W_L \approx C + h_0 L \tag{14}$$

where C is some constant we don't care about. So if we regress $\widehat{W}_L$ on L, the slope will be an estimate of h_0. Call this the *regression* estimate.

The third way is to use the fact, mentioned in the on-line notes about the topological entropy rate, that

$$h_0 = \lim_{L\to\infty} \log W_L - \log W_{L-1} \tag{15}$$

(Notice that this also follows from the linear expression I gave above.) So yet another estimate of h_0 is to take

$$\log \widehat{W}_L - \log \widehat{W}_{L-1} \tag{16}$$

for some large L. Call this the *difference* estimate.

All three estimates will ultimately converge on the same value, if you feed them enough data. In principle, all of them work best

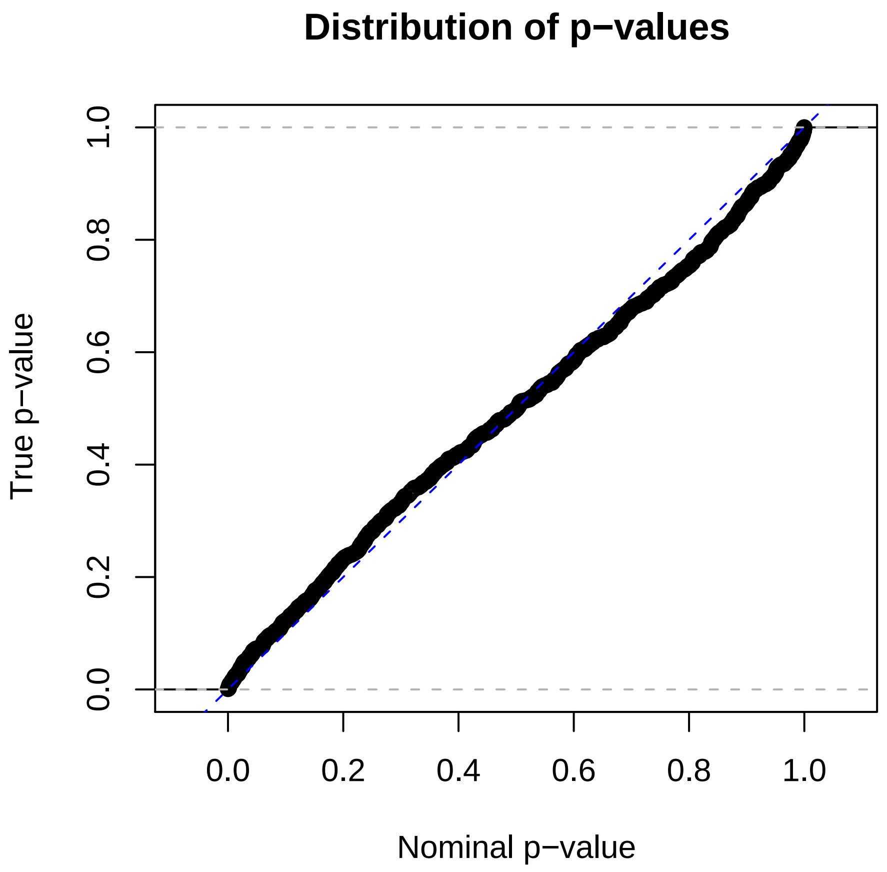

Figure 2: Distribution of p-values obtained from question 2 (black circles), with theoretical uniform distribution (dashed blue line).

when the value of L is large. In practice, if L is too large relative to n, we see only a very small sample of the allowed words, i.e., $\widehat{W}_L$ becomes much smaller than W_L, introducing systematic errors into our estimate. At the very least, when $\widehat{W}_L > \widehat{W}_{L+1}$, we do not have enough data to say what is happening with the longer words.

For the logistic map, we use a binary alphabet (symbols set), so there are at most 2^L words of each length. To give us some chance of seeing each of them, we should use a symbol sequence which is at least a few times longer than the number of words we might run into, say 10×2^L.

Here's how to do the division estimate.

```
logistic.TER.division <- function(r,L,n=10*(2^L)) {
  s <- logistic.symbseq(n,r)
  blocks <- symbseq.to.blocks(s,L)
  word.table <- table(blocks)
  W.L <- dim(word.table) # Counts number of distinct allowed words
  return(log(W.L)/L)
}
```

Notice the trick with using the `table` function to identify all the *distinct* words. Let's re-cycle that for the regression estimate.

```
logistic.TER.regression <- function(r,L,n=10*(2^L)) {
  s <- logistic.symbseq(n,r)
  W <- vector(mode="numeric",length=L)
  for (i in (1:L)) {
    blocks <- symbseq.to.blocks(s,i)
    W[i] <- dim(table(blocks))
  }
  my.regression <- lm(logcounts ~ lengths,
                      data.frame(logcounts=log(W),lengths=(1:L)))
  return(as.vector(my.regression$coefficients[2]))
}
```

And here's the difference estimate:

```
logistic.TER.difference <- function(r,L,n=10*(2^L)) {
  s <- logistic.symbseq(n,r)
  lastW <- dim(table(symbseq.to.blocks(s,L)))
  nextotlastW <- dim(table(symbseq.to.blocks(s,L-1)))
  return(log(lastW) - log(nextotlastW))
}
```

To double-check these, notice that when $r = 1$, we have IID coin-tossing, and every sequence of length L is allowed, so $W_L = 2^L$. This means that h_0 should be $\log 2 = 0.6931472$.

```
> logistic.TER.division(1,3)
```

```
[1] 0.6931472
> logistic.TER.regression(1,3)
[1] 0.6931472
> logistic.TER.difference(1,3)
[1] 0.6931472
```

which checks out.

Finally, let's plot these estimates as functions of r to see if we're getting something reasonable. We know that h_0 should be zero whenever the logistic map goes to a limit cycle (see the online notes for details). I use $L = 6$ simply for reasons of speed. The plot is Figure 3.

```
> r.values <- seq(from=0,to=1,length.out=200)
> difference.values <- sapply(r.values,logistic.TER.difference,L=6)
> plot(r.values,difference.values,type="l",xlab="r",ylab="Estimated h0",
       main="topological entropy rate estimates")
> division.values <- sapply(r.values,logistic.TER.division,L=6)
> lines(r.values,division.values,lty=2)
> regression.values <- sapply(r.values,logistic.TER.regression,L=6)
> lines(r.values,regression.values,lty=3)
```

(b) The easiest way to get a value for the standard error here is simply to re-run the estimator multiple times and take the standard deviation. This only captures the error associated with the fluctuations from one run of the simulation to another, rather than the systematic errors which come from biases in the estimator, etc.

4. (a) Basically all the parts needed for this were already assembled for solving problem 2. We need to estimate the 2×2 transition matrix P_{ij}, and the maximum likelihood estimates are

$$\widehat{P}_{ij} = \frac{N_{ij}}{\sum_{j=0}^{1} N_{ij}} \tag{17}$$

where N_{ij} counts the number of times the symbol i is followed by the symbol j. We can get this from the `symbseq.to.successive.blocks` function we wrote, followed by using `table`.

```
markov.mle.1 <- function(s) {
  # Estimate a binary Markov chain
  blocks <- symbseq.to.successive.blocks(s,1)
  counts <- table(blocks)
  # prop.table converts a count table to proportions, either by rows
  # or by columns, depending on 2nd argument - see its help file
  mle <- prop.table(counts,1)
  log.like <- sum(counts[counts>0]*log(mle[mle>0]))
  return(list(transition.matrix=mle,log.like=log.like))
}
```

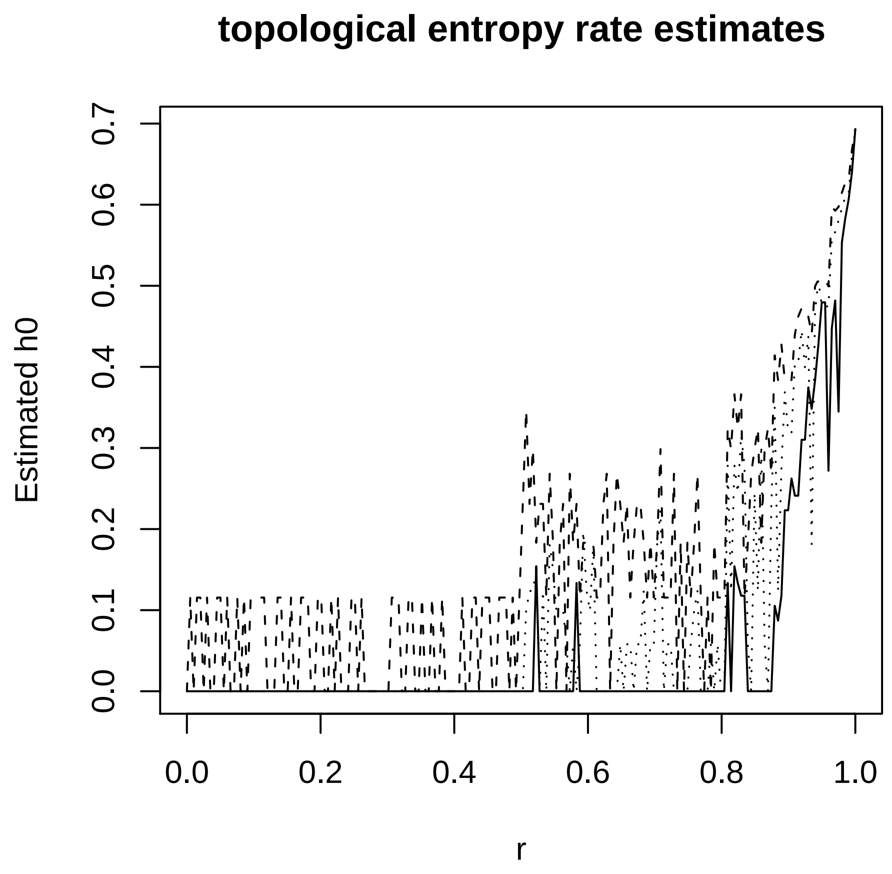

Figure 3: Three estimates of the topological entropy rate of the logistic map. Solid line, difference estimate. Dashed line, division estimate. Dotted line, regression estimate. The true value of h_0 is 0 whenever the map goes to a limit cycle, i.e., whenever $r < 0.866$ or so, suggesting that the division and regression estimates may have a larger upward bias than the difference estimate.

We'll need the log-likelihood in part c to compute the BIC scores, so we may as well compute it here, where it's easy. Remember it's

$$\mathcal{L} = \sum_{i,j} N_{ij} \log \widehat{P}_{ij} \tag{18}$$

Even though $0 \log 0 = 0$, `R` does not like taking log of 0, so the conditions in the sum restrict us to the strictly-positive terms.

Here's a quick check that this is working properly:

```
> ss <- rbinom(20,1,0.7)
> ss
 [1] 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 0 1
> table(symbseq.to.successive.blocks(ss,1))
       followers
leaders  0  1
      0  2  2
      1  2 13
> markov.mle.1(ss)
$transition.matrix
       followers
leaders         0         1
      0 0.5000000 0.5000000
      1 0.1333333 0.8666667

$log.like
[1] -8.662706
```

There are four positions where the sequence has "0"; two of them are followed by "0" and two of them are followed by "1". There are 15 positions where a "1" is followed by something. (There is also a last "1", at the end, and we don't know what it transitions to.) Of these, two of them are followed by "0". So the counts are correct, and converting the counts to probabilities gives us exactly what the function says. This isn't *right* — the data came from IID coin-tossing, with $p = 0.7$, so the right matrix would be

$$\left[\begin{array}{cc} 0.3 & 0.7 \\ 0.3 & 0.7 \end{array} \right]$$

but it's not crazy to not get this right with only 20 data points!

To see this program in a better light, let's write a small function to simulate from a binary Markov chain.

```
rbinmarkov <- function(n, p01, p11, p1start=-1) {
  # n is the length of the output.
  # A binary Markov chain has two free parameters in its transition
  # matrix, which can be chosen in several ways --- here the
  # probability that "0" is followed by "1", and that "1" is followed
  # by "1".
  # Also need the probability of starting in state 1 (since that
  # implicitly gives the probability of starting in state 0, as well)
  # The default negative number means that this is calculated from the
  # stationary distribution, below
  if (p1start < 0) {
    P = matrix(c(1-p01,p01,1-p11,p11),nrow=2)
    # The invariant distribution is the proportional to the
    # first eigenvector
    P.inv = eigen(P)$vectors[,1]
    # the eigen routine returns vectors whose norm is 1, so the
    # components don't sum to 1, so we need to fix that
    p1start = P.inv[2]/sum(P.inv)
    # We need the 2nd component because "0" is the first symbol.
  }
  s = vector(length=n)
  s[1] = rbinom(1,1,p1start)
  for (i in 2:n) {
    s[i] = rbinom(1,1,ifelse(s[i-1]<1,p01,p11))
  }
  return(s)
}
```

Now let's try our estimator on the output from this.

```
> rs <- rbinmarkov(1e4,0.3,0.7)
> markov.mle.1(rs)
$transition.matrix
       followers
leaders         0         1
      0 0.7086676 0.2913324
      1 0.3046236 0.6953764

$log.like
[1] -6088.478
```

Since the true matrix is $\begin{bmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{bmatrix}$, this seems to be working.

(b) We have most of the tools we need for this, too. We need to get the count for how often each block of length k is followed by each binary symbol (not by each block of length k again). We'll use `symbseq.to.blocks` twice.

```
symbseq.to.successive.blocks.varying <- function(s,L.lead,L.follow) {
  all.leaders <- symbseq.to.blocks(s,L.lead)
  all.followers <- symbseq.to.blocks(s,L.follow)
  n <- length(s)
  # Now need to pick only the leaders which actually have followers
  max.lead.index <- n - L.lead - L.follow +1
  leaders <- all.leaders[1:max.lead.index]
  # and pick the followers which have leaders, which is all but the
  # first L.lead of them
  followers <- all.followers[-(1:L.lead)]
  return(list(leaders=leaders,followers=followers))
}
```

We can test this by, first of all, checking that when $L_{lead} = L_{follow}$, it gives the same results as `symbseq.to.successive.blocks`.

```
> ss
 [1] 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 0 1
> symbseq.to.successive.blocks(ss,2)
$leaders
 [1] "11" "11" "11" "11" "11" "11" "11" "11" "11" "11" "10" "00" "00" "0
[16] "11" "11"

$followers
 [1] "11" "11" "11" "11" "11" "11" "11" "11" "10" "00" "00" "01" "11" "1
[16] "10" "01"

> symbseq.to.successive.blocks.varying(ss,2,2)
$leaders
 [1] "11" "11" "11" "11" "11" "11" "11" "11" "11" "11" "10" "00" "00" "0
[16] "11" "11"

$followers
 [1] "11" "11" "11" "11" "11" "11" "11" "11" "10" "00" "00" "01" "11" "1
[16] "10" "01"
```

Now for the estimator itself.

```
markov.mle.k <- function(k,s) {
  # Estimate a binary Markov chain of order k
  blocks <- symbseq.to.successive.blocks.varying(s,k,1)
  counts <- table(blocks)
  mle <- prop.table(counts,1)
  log.like <- sum(counts[counts>0]*log(mle[mle>0]))
  return(list(transition.matrix=mle,log.like=log.like))
}
```

(The backwards-seeming order of the arguments here is make things easier in part c.)

If this is working, then saying $k = 1$ should give us the same thing as `markov.mle.1`, which it does.

```
> markov.mle.k(1,ss)
```

```
$transition.matrix
       followers
leaders         0         1
      0 0.5000000 0.5000000
      1 0.1333333 0.8666667

$log.like
[1] -8.662706
```

But using a higher value of k should give us something else:

```
> markov.mle.k(2,ss)
$transition.matrix
       followers
leaders         0         1
     00 0.5000000 0.5000000
     01 0.0000000 1.0000000
     10 0.5000000 0.5000000
     11 0.1538462 0.8461538

$log.like
[1] -8.353788
```

(You can check by hand that this is the right transition matrix when $k = 2$.)

Similarly when we use a first-order Markov chain as an input, we should get something with (nearly) repeated rows in the output transition matrix:

```
> markov.mle.k(2,rs)
$transition.matrix
       followers
leaders         0         1
     00 0.7117615 0.2882385
     01 0.3015447 0.6984553
     10 0.7011417 0.2988583
     11 0.3057681 0.6942319

$log.like
[1] -6086.958
```

Notice that the log-likelihood has gone up, compared to the result with `markov.mle.1`.

(c) We just need to invoke `markov.mle.k` a few times:

```
markov.mle.bic <- function(kmax,s) {
  # Use BIC to estimate the order of a binary Markov chain (capped at
  # kmax), from a symbol sequence s
  n <- length(s)
  # The penalty in the BIC is (d/2)log(n), where d is the number of
  # free parameters
  # For a kth order Markov chain, with alphabet size m, there are
  # m^k states or contexts, and each of them has (m-1) transition
  # probabilities we need to estimate.  (See slides for lecture 6.)
  # Here m = 2, because we've got a binary alphabet.
  penalty <- function(k) { (2^k)*(1/2)*log(n) }
  # Estimate kth-order models for all k from 1 to kmax
  fits <- sapply(1:kmax,markov.mle.k,s)
  BIC <- as.numeric(fits[2,]) - penalty(1:kmax)
  best.k <- which.max(BIC)
  best.BIC <- max(BIC)
  # extra [[1]] in next line is de-referencing
  best.mle <- fits[1,best.k][[1]]
  best.loglike <- as.numeric(fits[2,best.k])
  return(list(k=best.k,transition.matrix=best.mle,log.like=best.loglike,
              BIC=best.BIC))
}
```

What we'd like is for this to pick the lowest possible order (1) for our running example (since that was from an IID source, which is really an order 0 Markov chain).

```
> markov.mle.bic(3,ss)
$k
[1] 1

$transition.matrix
       followers
leaders         0         1
      0 0.5000000 0.5000000
      1 0.1333333 0.8666667

$log.like
[1] -8.662706

$BIC
[1] -11.65844
```

With the first-order Markov chain example:

```
> markov.mle.bic(5,rs)
$k
[1] 1

$transition.matrix
       followers
```

```
leaders         0         1
      0 0.7086676 0.2913324
      1 0.3046236 0.6953764

$log.like
[1] -6088.478

$BIC
[1] -6097.689
```

So, again, even though the likelihood is higher at higher order, the BIC procedure successfully picks the lower-order chain. To really test this, I ought to feed it output from something which *is* a higher-order chain, but once again I'm just the teacher.

5. It will simplify things to have a consistent order in which to refer to the states. Which order doesn't matter, but I'll go from the top down in the figure in the Lecture 6 slides: "AAAB" = 1, "BA" = 2, "BAB" = 3, "BAA" = 4, "BAAB" = 5, "AAA" = 6, "BB" = 7.

   (a) This is a stochastic finite automaton, so at each time-step we have a state, which is hidden, and a symbol, which is what we want to output. The probability of each symbol depends on the state; the current state and the symbol fix the next state. There are several ways to do this, but one is to use two matrices, one for the symbol-emission probabilities, and one for the transitions.

   With a binary alphabet, the emission probability matrix can just be a vector, say the probabilities of producing the symbol "A". The transition matrix should have two columns, one for each symbol, and a row for each state; the first column gives the state gone to on "A", the second on "B".

```
foulkes.emit.A <- c(0.1875,0.5625,0.250,0.5625,0.750,0.1875,0.9375)
foulkes.trans <-  rbind(c(2,7),c(4,3),c(2,7),c(6,5),c(2,7),c(6,1),
                        c(2,7))
```

   To simulate, we pick an arbitrary starting state, then emit either an "A" or a "B" with a probability given by `foulkes.emit.A`, and then pick a new state according to `foulkes.trans`.

```
rfoulkes <- function(n,burn.in=0,debug=FALSE) {
  # Generate a symbol sequence of length n from the Foulkes process
  # in debug mode, return the state sequence as well as the symbol
  # sequence
  # burn.in gives number of steps to discard before beginning output
  # (used to approach stationary distribution via mixing)
  foulkes.emit.A <- c(0.1875,0.5625,0.250,0.5625,0.750,0.1875,0.9375)
  foulkes.trans <- rbind(c(2,7),c(4,3),c(2,7),c(6,5),c(2,7),c(6,1),
                              c(2,7))
  n.total <- n+burn.in
  x <- vector(length=n)
  if (debug) { recorded.s <- vector(length=n) }
  s <- sample(1:7,1)
  for (i in 1:n.total) {
    u = rbinom(1,1,foulkes.emit.A[s])
    emitted = ifelse(u == 1,"A","B")
    if (i > burn.in) {
      if (debug) { recorded.s[i-burn.in] <- s }
      x[i-burn.in] <- emitted
    }
    s = foulkes.trans[s,ifelse(emitted=="A",1,2)]
  }
  if (debug) { return(list(states=recorded.s,symbols=x)) }
  else { return(x) }
}
```

Notice the `burn.in` argument — it makes the process run for a certain number of steps, but only begins giving output after that burn-in is over. This is useful if we want to simulate from the invariant distribution of the process (so the data are stationary), but don't want to figure out what that is explicitly; we rely on mixing to figure it out for us! (In this case, it would be possible to construct the 7×7 transition matrix of the states and find then invariant distribution by taking its leading eigenvector, as in `rbinmarkov`, but that seems like more work, and the burn-in strategy is applicable when explicitly solving for the invariant distribution is impractical.)

Also notice the `debug` argument. When this is true, the function outputs the sequence of states as well as the sequence of symbols. This is useful for checking that the function is working properly. For instance, this checks that the symbol-emission probabilities are working right:

```
> prop.table(table(rfoulkes(n=1e5,debug=TRUE)),1)
      symbols
states          A          B
     1 0.18450646 0.81549354
     2 0.55959680 0.44040320
     3 0.25205819 0.74794181
     4 0.56649626 0.43350374
     5 0.75181422 0.24818578
     6 0.19366394 0.80633606
     7 0.93751516 0.06248484
```

and this checks that the only transitions which are happening are the ones which ought to be:

```
> table(symbseq.to.successive.blocks(rfoulkes(n=1e4,burn.in=100,
        debug=TRUE)$states,1))
       followers
leaders    1    2    3    4    5    6    7
      1    0  149    0    0    0    0  702
      2    0    0 1244 1516    0    0    0
      3    0  340    0    0    0    0  904
      4    0    0    0    0  665  851    0
      5    0  505    0    0    0    0  160
      6  851    0    0    0    0  227    0
      7    0 1766    0    0    0    0  119
```

We can also use the counts to check that states are working properly, e.g. that sates 2 ("BA") and 7 ("BB") are working:

```
> prop.table(table(symbseq.to.successive.blocks.varying(rfoulkes(1e4,
             burn.in=100),2,1)),1)
       followers
leaders          A          B
     AA 0.41663586 0.58336414
     AB 0.35635562 0.64364438
     BA 0.58008817 0.41991183
     BB 0.94753921 0.05246079
```

You can conduct similar tests for histories ("leaders") of length 3 and 4, picking out those states, and length 5 or longer (confirming that additional historical information doesn't change the conditional distributions). Note that to get accurate estimates of probabilities over long words (e.g., length 5 if $k = 4$) you need increasingly large simulations.

(b) The Foulkes process is can be written as a 4^{th} order Markov chain, with $2^4 = 16$ states, and we know (from Guttorp, p. 72) that BIC is a consistent estimator of the order of Markov chains, so $\hat{k}$ should approach 4 eventually.

```
> foulkes.k.1e3 <- replicate(100,markov.mle.bic(6,rfoulkes(1e3))$k)
> mean(foulkes.k.1e3)
[1] 3.53
> sd(foulkes.k.1e3)
[1] 0.5213619
> foulkes.k.1e4 <- replicate(100,markov.mle.bic(6,rfoulkes(1e4))$k)
> mean(foulkes.k.1e4)
[1] 4
> sd(foulkes.k.1e4)
[1] 0
```

so somewhere between $n = 10^3$ we go from estimating $\widehat{k} = 3.5 \pm 0.5$ to essentially always estimating $\widehat{k} = 4$. With this much data, too, the estimated probabilities are (usually!) within ± 0.05 of the true probabilities.