7. Control Structures

Z620: Quantitative Biodiversity, Indiana University

OVERVIEW

In Dante's satire of hell, each layer contains a new iterative task for the tortured to endure for eternity. Without a mastery of basic computational skills, wrestling with large data sets can make it seem like you are trapped in an underworld. Much of the power of computing stems from the efficiency of computers in performing small tasks in an iterative manner. This lesson introduces the concepts of control structures. We will use these tools to understand the diversity of jellybeans flavors, but control structures will be useful for automating many of your future work flows in a reproducible manner.

1) INTRODUCTION TO CONTROL STRUCTURES

Control structures allow you to control the flow of execution in a script. These blocks of code serve as algorithms that act on variables in user-defined way. Often, control structures are embedded in functions to carry out sequential, selective, and repetitive tasks. Some of the most commonly used control structures include "if, else" statements, "while" loops, and "next" statements. Today, we will focus mostly on the "for loop".

There is a lot of information on control structures elsewhere. A succinct tutorial of common control structures in R can be found at https://ramnathv.github.io/pycon2014-r/learn/controls.html You can also take a look at `?Control` for documentation.

```
rm(list = ls()) # clear working directory
```

A. 'If, Else' Logical Tests

These control structures are useful for making decisions. The 'if' statement evaluates a condition. Based on the outcome of that evaluation, an 'else' statement can be used to execute a specific code. The following text will walk you through how to use and implement 'if, else' control structures.

1. The general syntax for an "if, else" control structure in the R language: if (condition) {
   # do something
   } else {
   # do something else
   }
2. An example of the an "if, else" control structure.

```
x <- 1:15
if (sample(x, 1) <= 10) {
   print(paste("x is",x ,"and is less than 10"))
} else {
   print(paste("x is",x,"and is greater than 10"))
}
```

3. "if" accepts logical values (e.g., <, >=, !=); can accept numbers but not strings.
4. "if" can be complemented by an "else"
5. `ifelse()` is a function that can be used in R (e.g., ifelse(test_expression, x, y))
6. if-else can be strung to create hierarchy of tests

B. For-Loops

These control structures execute an iterative task. For-loops are made up of two basic parts. The first part, sometimes referred to as the header, specifies the iteration and has a loop counter. The second part of the for-loop is the body, which contains an operation that is executed upon each iteration.

Syntax The specific syntax of for-loops will look different in other languages but basic syntax and structure remains the same.

1. Start a loop with for

2. Declare variable and number of iterations

3. Tasks to perform in code block (or "body")

```
1.      2.          3.
```

In R this looks like: for(i in 1:10){ print(i) }

```
for(i in 1:10){
  print(i)
}
```

In the above for-loop structure, i is a variable that will hold each integer from 1 to 10 during each successive pass through the loop.

You can use this structure to move through each item of vector and perform any number of tasks to each element.

For example, let's print out a list of fruits.

```
x <- c("apples", "oranges", "bananas", "strawberries") # x is a vector of fruit names

for (i in seq(x)) {
    print(x[i])
}
```

Note that a for-loop can iterate over any type of vector: logical, numerical, string. You can also iterate over vectors using indices (R counts from 1 and not from 0 like some other languages) or assign i as a local variable that holds the value of each item in a vector.

```
for (i in x) {
    print(i)
}
```

You can use for-loops to transform vectors. The two scripts below give use the length of the strings in our vector. However, the first one prints to the screen a local scalar variable, `len`, and the second uses the loop to fill a global scalar vector, `len`, that now holds the length of each string.

```
for (i in seq(x)) {
  len <- nchar(x[i])
  print(len)
}

len <- ""
for (i in seq(x)) {
  len[i] <- nchar(x[i])
}
print(len)
```

Note on the use of for-loops
While for-loops are powerful and commonly used, there are some downsides. They take a long time to execute and are somewhat clumsy. In R, there are some other ways to achieve the same power. For example, the function `tapply()` executes a function to specific elements of a data frame, `apply()` executes a function (e.g., mean) to the rows or columns of a data frame, and `sapply()` performs a function on list or array.

These functions are easier to implement and are also more computationally efficient. Check out `?tapply`, ?apply`,`?sapply‘.

```
len <- sapply(x, nchar)

#the output of sapply is a "named int", but this can by easily coerced to a vector.
len <- as.vector(len)
print(len)
```

C. Combining control structures: ‘for’ and ‘if’

Control structures can be combined. For example, if we only want to print numbers that can be divided by 3, we can use the modulus operator (%%) to get calculate the remainder:

```
for(i in 1:10){
  if(i %% 3 == 0){
    print(i)
  }
}
```

D. While loops

Another loop control structure is the “while loop”. While loops are sort of like repeating “if” statement; that is, it continues to operate until a conditional statement changes (e.g., “true” becomes “false”).

Here is an example of a while loop.

```
i <- 1
while (i < 10) {
    print(i)
    i <- i + 1
}
```

Notice that a while loop needs an specific break point.

What would happen if the code block was changed to subtract 1 from i in each iteration of the loop?
If you dare, try this out!

2) APPLYING CONTROL STUCTURE TO DIVERSITY DATA

The control structures discussed above have unlimited applications in the realm of quantitative methods. In this section you will learn how to apply these structures to biodiversity data, specifically the jelly bean communities that you just characterized. But before we can do that we need to get familiar with another powerful family of functions in R: built-in probability distributions.

A. Statistical Distributions

Many groups of quantities can be described as frequency distributions. You are probably familiar with the Normal distribution and its hallmark bell curve. R has built-in functions describing many statistical distributions. You can find them listed in the R Documentation by typing `?Distributions` in the console. Choose your favorite distribution and look at its specific documentation. You will find that each distribution has four functions associated with it: **d**ensity, **p**robability, **q**uantiles and **r**andom sample. For example

the uniform distribution (an equal probability for all values) is called *unif* in R and has the following four functions.

```
dunif(x, min = 0, max = 1, log = FALSE)                          # Density
punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)     # Probaility
qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)     # Qunatile
runif(n, min = 0, max = 1)                                       # Random Sample
```

A Brief Note on Random Sampling by Computers
Computers can't generate truly random number, but rather pseudo-random numbers. This is achieved by a function that generates sequences of numbers that behave like random numbers, but are completely determined by the functions input known as a *seed*. Put in the same seed twice and you will get the same sequence of numbers.

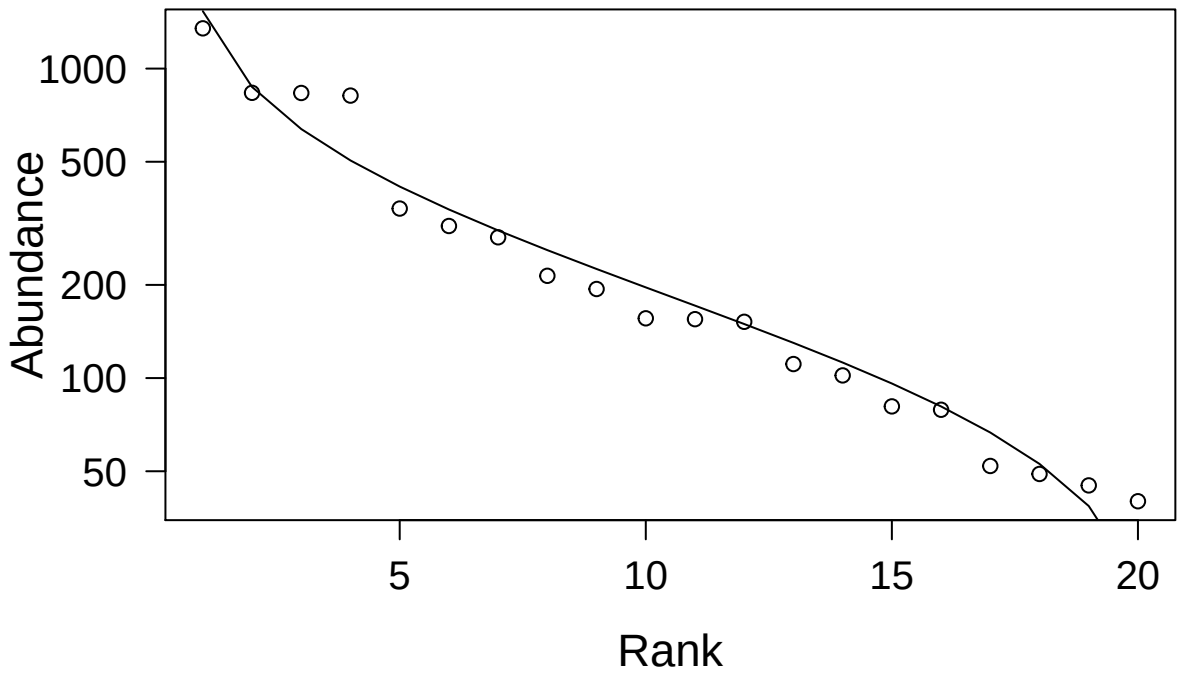
B. Generating the Source Community

In the Diversity Sampling exercise, we used jelly beans to simulate a biological community. Here, we will simulate a digital community assuming that species abundances follow a log-normal distribution. We will draw a sequence of random numbers from the log-normal distribution. Each of these numbers will represent the abundance of a single species in our simulated community.

```
# for reproducible results we will set the seed
set.seed(6)

# draw 20 random numbers from log-normal distribution with a mean of 5 (log scale)
jelly.jar <- rlnorm(n = 20, meanlog = 5 )

# force numbers to integers
jelly.jar <- as.integer(jelly.jar)
```

Let's plot the rank-abundance curve of our community using the `rad` function from the `vegan` package

```
require (vegan)
RACresults <- rad.lognormal(jelly.jar)
plot(RACresults, las = 1, cex.lab = 1.4, cex.axis = 1.25)
```



B. Sampling the Source Community

Next we will write a function that will draw a random sample from our `jelly.jar` community. Take a moment to break down the process of sampling into a sequence of actions.

One such sequence may be:
1. draw random jelly bean
2. classify jelly bean color
3. add 1 to list of colors by color sampled
4. repeat steps 1-3 N times, where N is your sample size

This sequence of actions is an algorithm. Here is how looks in a user-defined function:

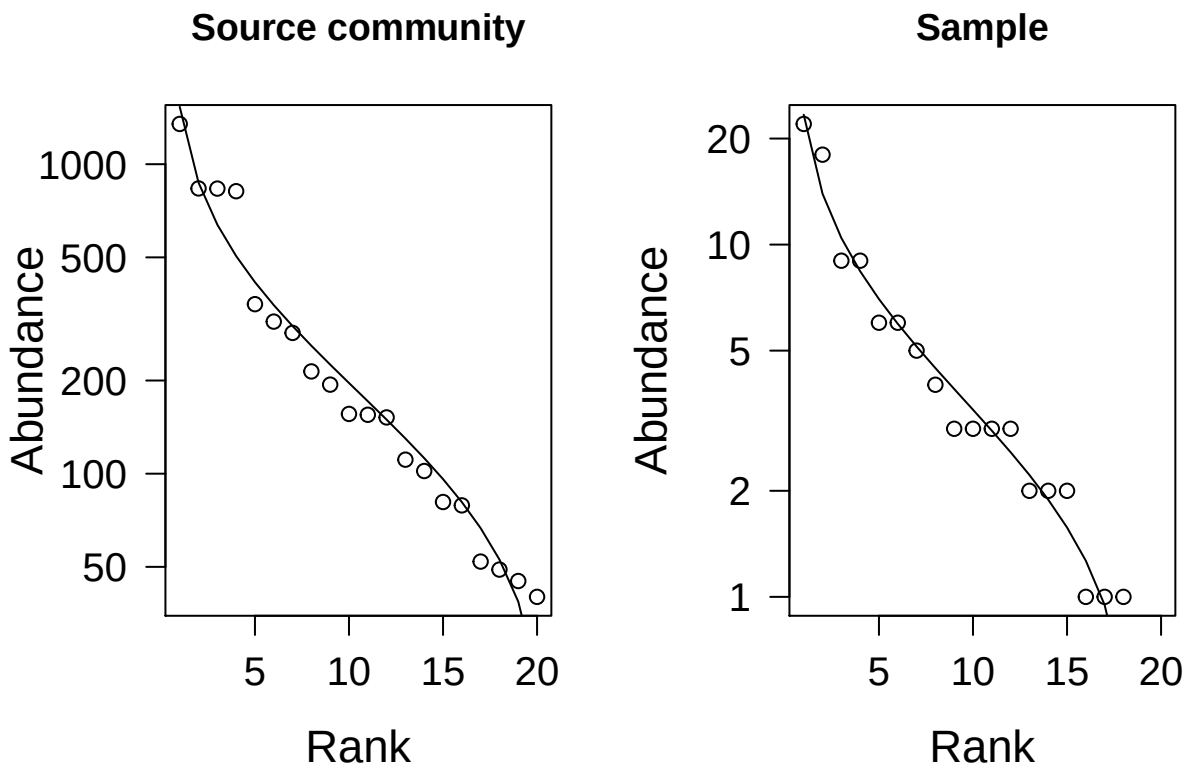
```
sample.community <- function (x,n){
  # write out an explicit vector of the community
  all.individuals <- rep(seq(x), x)

  # take n samples from the community
  survey <- sample(all.individuals,n)

  # prepare vector of species with 0 in each bin
  survey.sum <- rep(0, length(x))

  # add 1 to species bin each time an individual of that species was sampled
    for ( i in survey){
     survey.sum[i] <- survey.sum[i] + 1
    }
```

```
  return(survey.sum)
}
```

Note that we drew multiple samples using the `sample` function. This function iterated over the sampling step `n` times. Next we wrote a for-loop to iterate over those samples and bin them into our species list.

Let's compare the rank abundance curves of a sample to our source community.

```
# for reproducible results we will set the seed
set.seed(3)

# sample using our fuction
sample.jelly <- sample.community(x=jelly.jar,n=100)

# generate RAC using vegan
RACsample <- rad.lognormal(sample.jelly)
plot.new()
par(mfrow=c(1,2))
plot(RACresults, las = 1, cex.lab = 1.4, cex.axis = 1.25, xlim=c(1,20), main="Source community")
plot(RACsample, las = 1, cex.lab = 1.4, cex.axis = 1.25, xlim=c(1,20), main = "Sample")
```



Question: How similar or different is the sample from the source community?

Testing input to a function using the `if` statement

Functions work very well if they are fed the right kind of input. But pass some inappropriate argument and things will go wrong, and possibly horribly wrong. To demonstrate this idea try and sample the `jelly.jar` with a sample size larger than the size of the community.

```
sample.community(x=jelly.jar,n=(sum(jelly.jar)+1))
```

R produces an error message:

> Error in sample.int(length(x), size, replace, prob) : cannot take a sample larger than the population when 'replace = FALSE'

The first line of this message indicates the function that produced it. In this case it is the `sample.int()` function. This function is nested within the `sample()` function (you can see the code for `sample` by typing *sample* without brackets in the console). The second line of the error message tells us that you cannot take more sample the the population size unless you set the `replace` argument to TRUE. This message is very helpful.

To prevent such errors, that could lead to unexpected behavior of your code, it is a good idea to test the input of functions before processing it in the function. Let's rewrite our sampling function to include a test for sample size using the `if` statement. We will also use the `stop` function to produce our own error message.

```
sample.community <- function (x,n){

  # sample size cannot exceed community size
  if (n > sum(x)) stop ("Sample size cannot exceed total size of community")

  # write out an explicit vector of the community
  all.individuals <- rep(seq(x), x)

  # take n samples from the community
  survey <- sample(all.individuals,n)

  # prepare vector of species with 0 in each bin
  survey.sum <- rep(0, length(x))

  # add 1 to species bin each time an individual of that species was sampled
    for ( i in survey){
     survey.sum[i] <- survey.sum[i] + 1
    }
  return(survey.sum)
}
```

Now if we pass to our function `n` that exceeds the community size we will get an error from our function.

```
sample.community(x=jelly.jar,n=(sum(jelly.jar)+1))
```

> Error in sample.community(x = jelly.jar, n = (sum(jelly.jar) + 1)) : 'Sample size cannot exceed total size of community

Of course, there are many ways by which a functions arguments can be wrong, to rephrase Tolstoy *"All happy functions are alike; each unhappy function is unhappy in its own way"*. We demonstrated a logical error. Another common error is passing data of the wrong type to a function, such as a string when a number is called for. These errors can be caught using the *is.type* functions, where *type* is replaced by the type of interest (e.g.`is.numeric`). While We cannot prevent all errors, when writing a function it is worth taking a moment to think about how to prevent some common errors.

C. Estimating Richness via Resampling

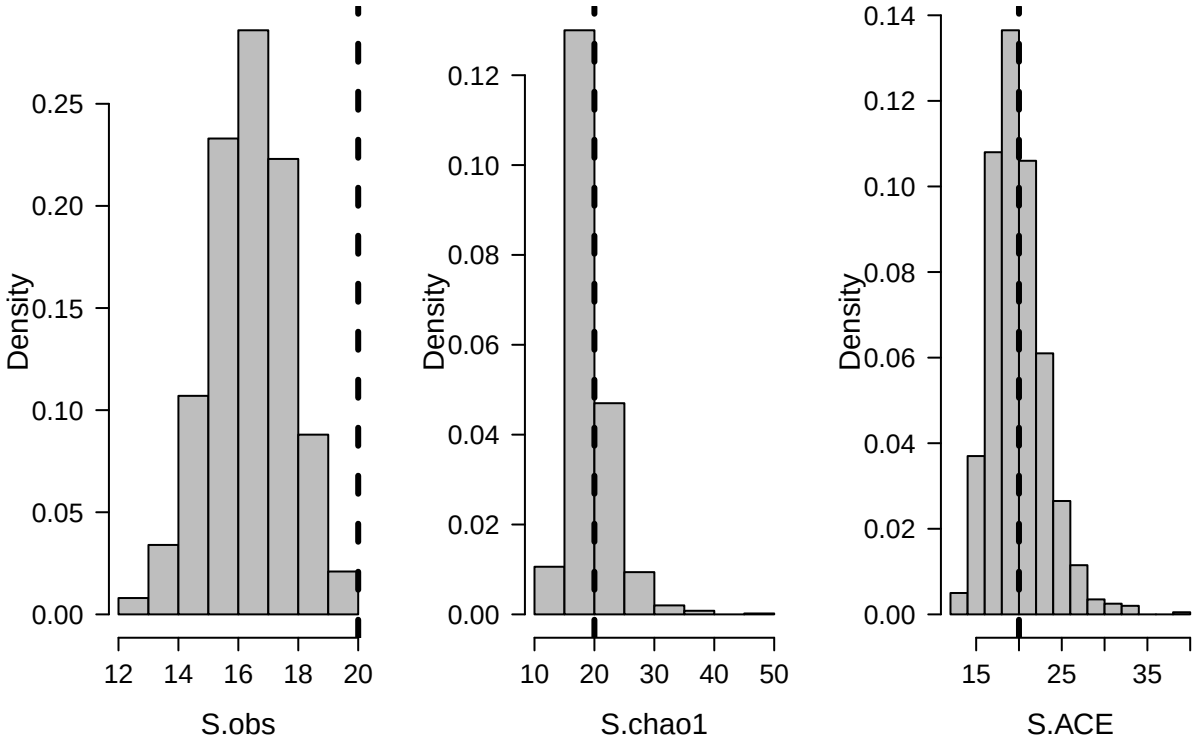
We will next use for-loops to explore how well richness indices succeed in estimating the true richness of the source community. We will use the function `estimateR()` from the `vegan` package to calculate diversity.

```
S <- estimateR(sample.jelly)
S
```

```
##     S.obs   S.chao1  se.chao1     S.ACE    se.ACE
## 18.000000 18.750000  1.417910 19.680642  1.722197
```

How well do these three indices estimate the true diversity? Using our simulated community we can test that! We will draw repeated samples from our source community and estimate S for each of them. Last, we will look at the distribution of the results compared to the true S.

```
# Number of individuales per sample
n <- 100

# Number of repeated samples
N <- 1000

# initialize matrix to store results
repeated.S <- matrix(NA, nrow = N, ncol = length(S))
colnames(repeated.S) <- names(S)

# resample and store results
for (i in seq(N)){
  sample.jelly <- sample.community(x=jelly.jar,n=n)
  repeated.S[i,] <- as.vector(estimateR(sample.jelly))
}

#plot distribution of results as histogram
plot.new()
par(mfrow=c(1,3))
for (i in c("S.obs", "S.chao1", "S.ACE")){
  hist(repeated.S[,i],prob=TRUE,  cex.lab = 1.4, cex.axis = 1.25, xlab = i,
  main = NULL, col = "grey", las = 1)

  #add dashed line to mark true S
  abline(v = length(jelly.jar), lty = 2, lwd = 3)
}
```



Play around with the sample size (by changing the value of `n`). How does sample size affect observed and estimated richness?

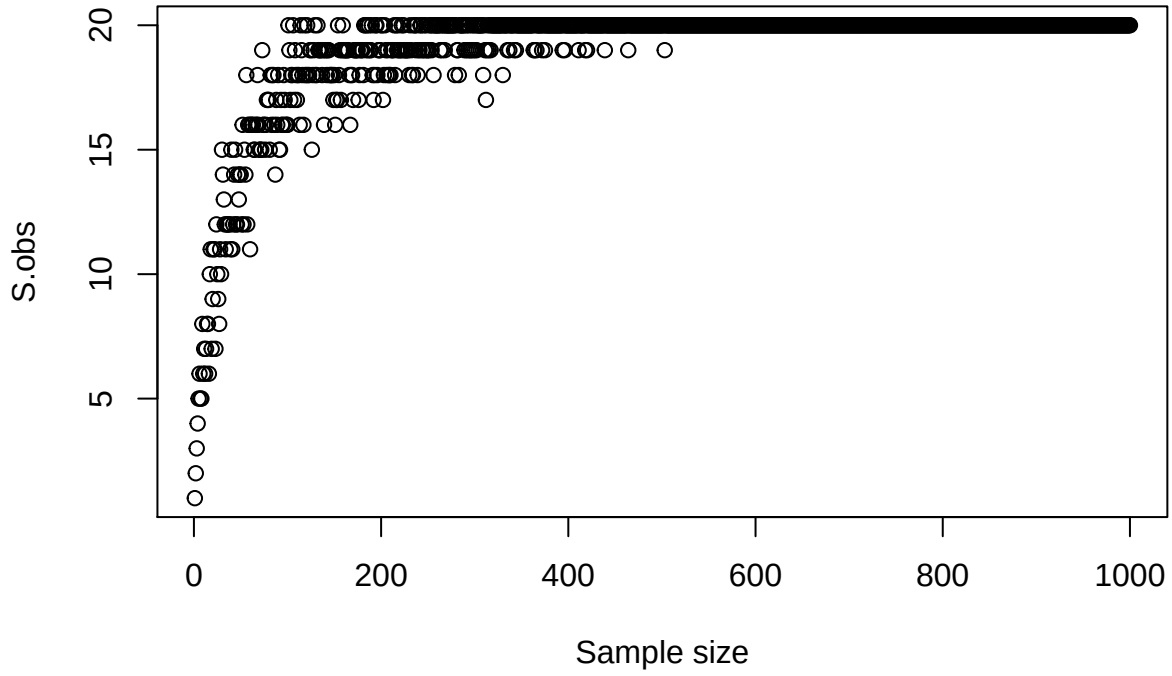
E. Collector's Curve

A collectors' curve is similar in ways to the rarefaction exercise we introduced in **4. Alpha Diversity**. Also known as the species accumulation curve, the collector's curve plots the cumulative number of species (i.e., richness) as a function of sampling effort (i.e., number of individuals sampled, N). The shape of the resulting curve can be informative. Qualitatively and practically, one can assess the amount of time and effort it will take to characterize the diversity of a sample. Quantitatively, data from the collector's curve can be fit with mathematical functions, and this information can be used to infer other biodiversity patterns, such as the species area relationship (SAR), which is discussed in **5. Spatial Diversity**.

In the code chunks below, we will first take a single sample of size N from the source community. We will then use for-loops to sample the source community with increasing effort (i.e., $N + 1$) while recording sample richness:

```
#take a single sample of size N from the source community
N<-1000
sample.jelly <- sample.community(jelly.jar,n = N)

#generate vector to store results
S.collectors <- rep(NA, N)

#loop through 1:N and check S.obs for sub-samples
for (i in 1:N){
  sub.sample <-  sample.community(sample.jelly,n = i)
```

```
  S.collectors[i] <-estimateR(sub.sample)["S.obs"]
}
# Plot results
plot.new()
par(mfrow=c(1,1))
plot(1:N, S.collectors, xlab = "Sample size",ylab = "S.obs")
```



Given that resources (time and money) are often limiting, what sample size would you use for estimating richness of the source community?