Efficiency Tips for Basic R Loop

Svetlana Eden, Department of Biostatistics
Vanderbilt University School of Medicine

June 4, 2012

Contents

1 Efficiency Defined. **2**

2 The Flexibility of R. **2**

3 Vector Operations and Their Time Complexity. **4**

4 Regular loop by-index and it's complexity. **6**

5 Myth Busters: sapply() and its efficiency **8**

6 Loops with quadratic time complexity. **9**
6.1 Reguar loops by-name and their complexity 10
6.2 Loops with c(), cbind(), rbind() 13
6.3 Loops and subsets 15

7 Summary **15**

8 Example **15**
8.1 Single record per ID 15
8.2 Multiple records per ID 17

9 Aknowledgements. **22**

10 Appendix. Example Code. **23**

List of Figures

1 5
2 7
3 8
4 12
5 14

List of Tables

1 Efficiency Defined.

Code1 and Code2 (see below), perform the same computational task. Which code is more efficient ? It's hard to say without defining efficiency first.

Code 1.

```
uniqueIDs = unique(data$id)
for (bi in 1:bootTimes){
  bootstrappedIDs = sample(x=uniqueIDs, size=length(uniqueIDs), replace=TRUE)
  bootstrappedData = data[data[[idVar]]==bootstrappedIDs[1],]
  for (i in 2:length(bootstrappedIDs)){
    bootstrappedData = rbind(bootstrappedData, data[data$id==bootstrappedIDs[i],])
  }
res = bootstrappedData
```

Code 2.

```
indPerID = tapply(1:nrow(data), data$id, function(x){x})
lengthPerID = sapply(indPerID, length)
for (bi in 1:bootTimes){
  bootstrappedIDIndices = sample(x=1:length(indPerID), size=length(indPerID), replace=TRUE)
  newDataLength = sum(lengthPerID[bootstrappedIDIndices], na.rm=TRUE)
  allBootstrappedIndices = rep(NA, newDataLength)
  endInd = 0
  for (i in 1:length(bootstrappedIDIndices)){
    startInd = endInd + 1
    endInd = startInd + lengthPerID[bootstrappedIDIndices[i]] - 1
    allBootstrappedIndices[startInd:endInd] = indPerID[[bootstrappedIDIndices[i]]]
  }
  res = data[allBootstrappedIndices, ]
}
```

For some, Code1 is more efficient because it's shorter (we spent less time typing it). Or, if we are more interested in readability of the code we will rate them both as inefficient because of the tiny font. In the context of the talk, we agree that given that two pieces of code perform the same task, one code is more efficient than the other if it runs faster or uses less CPU time.

Durting the talk we will evaluate the efficiency of different coding constructs in R. We will familiarize ourselves with the concept of time complexity – a measurement of efficiency. Using this concept, we will compare the performance of several R loops, pointing out which coding constructs should be avoided to maximize efficiency. We will also look at an example of inefficient code and will work through how to significantly improve its efficiency.

2 The Flexibility of R.

When we work with R, we have to choose a certain data structure. Our choices are:

- vector
- data.frame
- matrix
- list
- array

The choice of the data structure is usually straight forward. For example, when we have a list of subject IDs, we can store it in a vector. When we have several subjects, and each one has a set of records, we can use a data.frame. When we need to store a variance-covariance matrix, we could use a data.frame, but a matrix would be a better choice

because matrices come with convenient and fast matrix operations. And when we want to store everything in one object, the only data structure that can do this is a list.

So inspite of many choices choosing a data structure in R is intuitive and easy. Having many choices may also be confusing. Suppose we are using a data.frame.

```
> data

   id val
1 AAA 112
2 BBB  56
3 CCC  99
```

We would like to retrieve some data from it. More specifically, we want to get an element in the first row and in the first column, "AAA". Retrieving an element from a data structure is called "referencing". One way of referencing using the first column name ("id") and the first row index (1) is :

data$id[1]

Because of R's flexibility we can accomplish the same task in multiple ways including:

- `data$"id"[1]`

  the same as the previous one but with quotes. Like the previous way, it doesn't allow to pass variable "id" as an argument to a function.

- `data[["id"]][1]`

  allows passing variable "id" to function, the same type of referencing, but too many brackets.

- `data[1, "id"]`

  same as the previous one, but less brackets (and therefore typing)

- `data["1", "id"]`

  referencing a row by name

- `data["1", 1]`

  referencing a row by name, and a column by index

- `data[[1]][1]`

  referencing a row and a column by index, with some extra brackets

- `data[1, 1]`

  same but less brackets (less typing)

With all these choices at hand, it is our responsibility to choose the one that fits our needs. In the context of this talk, the way of referencing, by index or by name, makes a big difference. Referencing by name is provided by R to make our lives easier, because we remember names better than numbers. Indices are better for the computer. If the

computer knows the index, it can calculate the address of the data element and can access it very fast without searching. It takes longer for the computer to search for a name. If we compare different ways of referencing one element, we might not feel the difference, but the larger the data is, the more noticeable the difference becomes.

3 Vector Operations and Their Time Complexity.

To process large data we need a loop. Suppose we have a vector:

```
> vec = rep(1, 10)

> vec

[1] 1 1 1 1 1 1 1 1 1 1
```

And we would like to multiply each element by 2. One way of doing this is:

```
> vec*2

[1] 2 2 2 2 2 2 2 2 2 2
```

In this example we did not reference each element of the vector. Performing arithmetic operations on a vector as a whole (without referencing each element) is called a vector operation. Vector operations might not look like it, but they are loops written in C, and hidden by R. They are the fastest loops in R. In order to show this we present a graph with X-axis displaying the length of the vector, and Y-axis displaying how much CPU time is spent on this vector operation:

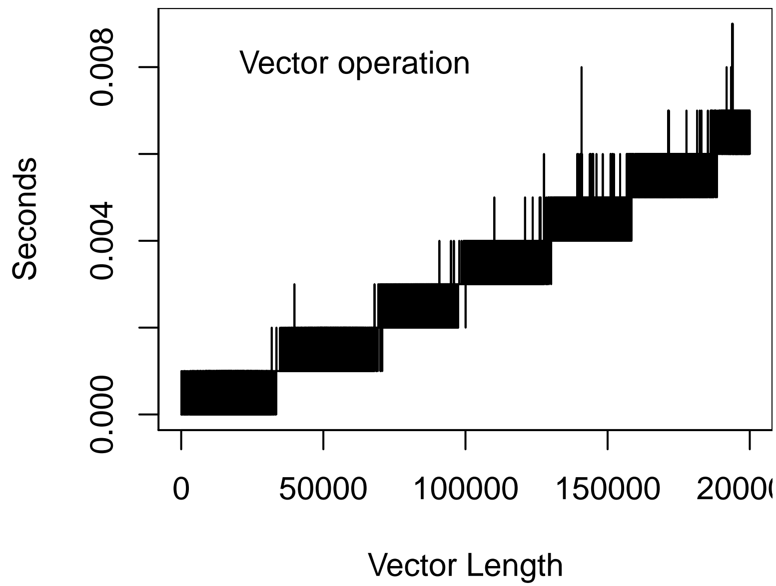


Figure 1: Efficiency of vector operation as a function of vector length

The graph shows CPU time as a function of the vector length. This function is monotonic: the longer the vector, the longer it takes to complete an operation on it. Note that the function is approximately linear. We will refer to this function as time complexity.

To show that vector operations are the fastest loops in R, we compare their performance to other loops. Learning other ways of running a loop is important since vector operations cannot do everything. For example, if you try to compare $(2.4 - 2.1)$ to 0.3 you will see the following:

```
> (2.4 - 2.1)
[1] 0.3
>
> (2.4 - 2.1) == .3
[1] FALSE
```

This is a known problem of binary number representation. One way of handling it is:

```
> isTRUE(all.equal(2.4-2.1, 0.3))

[1] TRUE
```

This will not work on vectors unfortunately, so to implement a similar comparison for a vector we have to use a regular loop.

4 Regular loop by-index and it's complexity.

Here is an example of a regular loop:

```
vec = rep(1, n)

for (i in 1:n){
  vec[i] = vec[i]*2
}
```

Let's graphically compare time complexity of the regular loop and the vector operation:

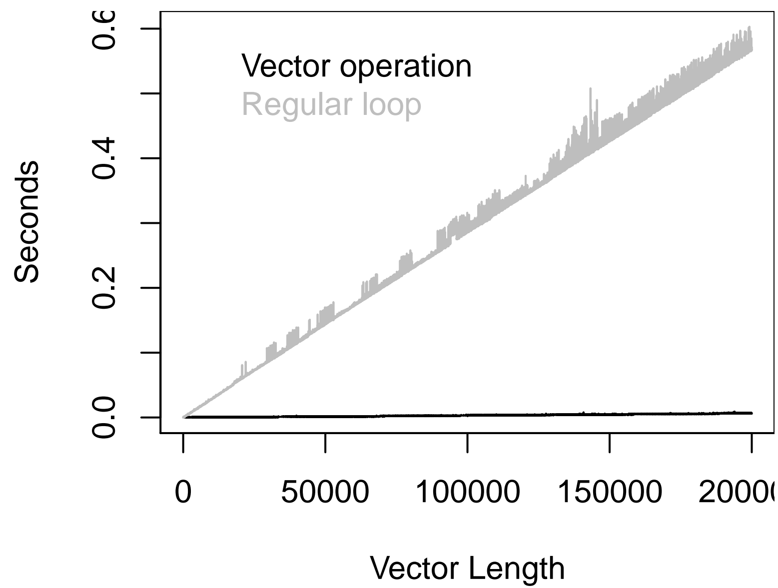


Figure 2: Graphical comparison of efficiency of vector and regular loop

As the graph shows, the performance of the vector operation is much better. We also notice that the regular loop's time complexity is approximately linear, similar to the vector operation. We will denote linear time complexity as $O(n)$.

5 Myth Busters: sapply() and its efficiency

R has many ways of making programming easier. One of them is function sapply(). It can be used with different data structures and plays a role of a regular loop.

```
> vec = c(10, 20, 30, 40, 50)

> sapply(vec, function(x){x*2})

[1]  20 40 60 80 100
```

There is a myth among R users that sapply() when applied to a vector is faster than a regular loop. To check if this is true let's look at the graph.

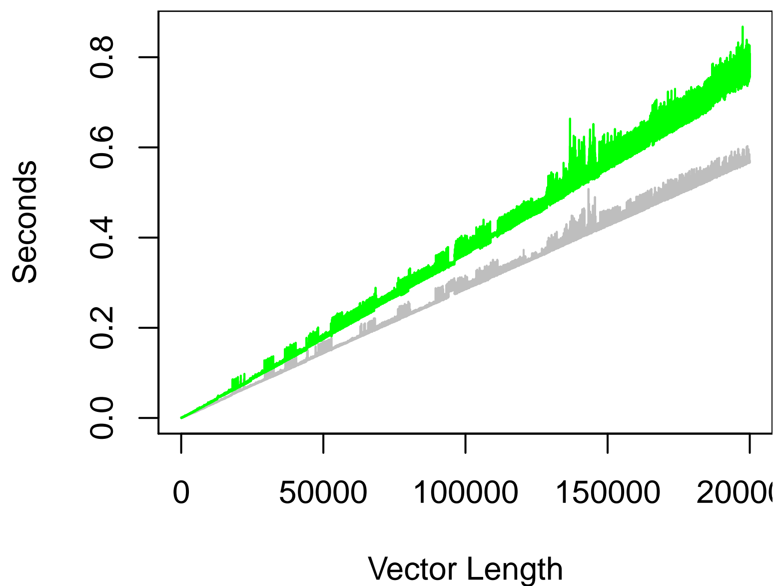


Figure 3: Graphical comparison of efficiency of regular loop and sapply()

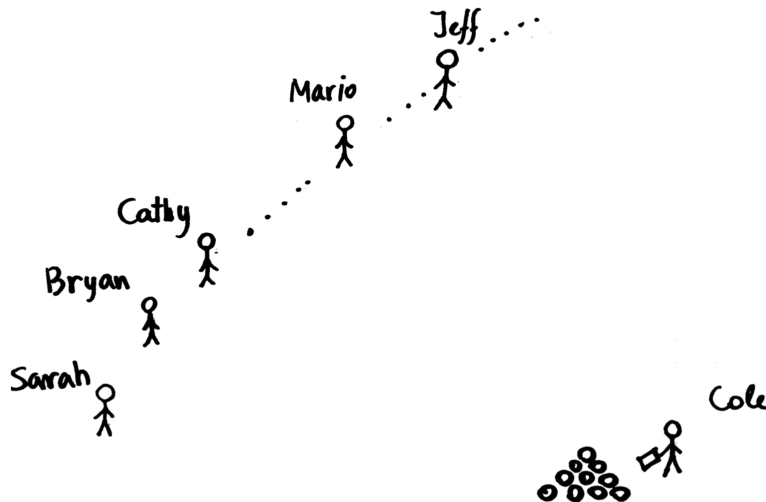
The myth about sapply() is dispelled by the graph, where the upper plot is the time of sapply() and the lower plot is the time for the regular loop. The victory is small, but the regular loop is still more efficient.

6 Loops with quadratic time complexity.

So far we have seen loops with linear complexity. We learned that:

- vector operations are the fastest, but cannot perofrm all operations
- regular loops by-index are slower than vector operations, but they it can do everything
- sapply() applied to vectors doesn't improve efficiency

Now let's talk about less efficient loops. The following example will help us understand why they are less efficient. As you might know, our department has a softball team. Let's imagine that at the end of each season our team holds a closing ceremony, during which our coach distributes gameballs to each player. The balls are named and distributed in the following way. The coach, Cole, is standing on one side of the field and the rest of the team stands on the other side.



The team is so far away from the coach that he cannot see the faces, but he knows what player is standing where because he has a list with each player's name and his/her place in the line:

- Cathy - 3rd
- Mario - 8th
- Bryan - 2nd
- Sarah - 1st
- Jeff - 11th
- and so on ...

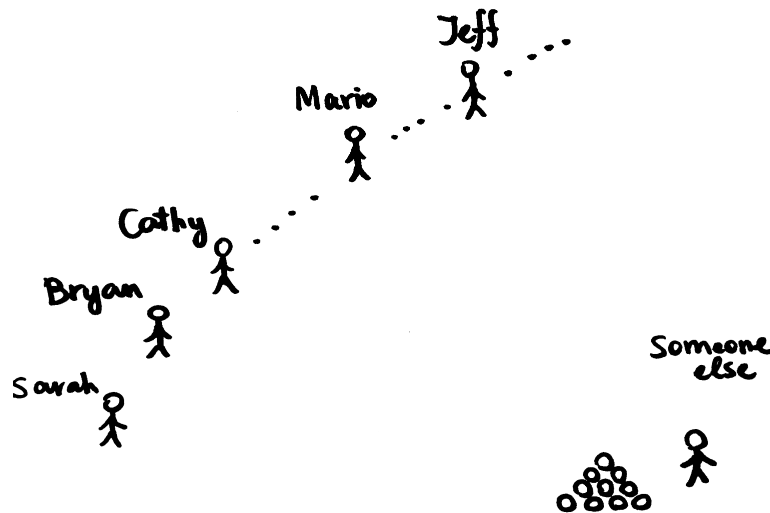
Let's calculate the complexity of this ceremony, but instead of time we will use number of throws made by the coach. Since Cole knows the players' places, the complexity is linear since

- Cathy is 3rd - one throw
- Mario is 8th - one throw
- Bryan is 2nd - one throw
- Sarah is 1st - one throw
- Jeff is 11th - one throw
- and so on

Therefore, the complexity $= 1 + 1 + 1 + 1 + 1 + ... = O(n)$

6.1 Reguar loops by-name and their complexity

Now, suppose that coach Cole could not make it to the ceremony. The league provided the team with a substitute coach. The substitute coach was not familiar with the ceremony. He knew that he had to distribute softballs, but didn't know that they were signed and he didn't have a list. Since he didn't know who is where, he decided to systematically



throw each ball to the first player in line. If the ball didn't belong to the player, the player would throw it back to the coach. For example, if the ball had Cathy's name on it, the coach would throw it to Sarah, and Sarah would throw it back to him. Then he would throw it to Bryan, and Bryan would throw it back. Then the coach would throw it to Cathy, and she would keep the ball. Next, if the ball had Mario's name on it, the coach would start with Sarah again, and he throw the ball to everybody in the line until it finally got to Mario. And so on. Let's calculate the complexity of this ceremony.

- Cathy is 3rd - three throws
- Mario is 8th - eight throws
- Bryan is 2nd - two throws

- Sarah is 1st - one throw
- Jeff is 11th - eleven throws
- and so on

Overall $3 + 8 + 2 + 1 + 11 + ... = 1 + 2 + 3 + 4 + ... = n(n + 1)/2 = O(n^2)$, the complexity is quadratic.

How is this example related to our talk ? As we mentioned at the beginning, R provides us with several ways of referencing data elements - by index and by name. We can also run a loop by index or by name. Earlier we saw an example of a loop by index:

```
vec = rep(1, n)

for (i in 1:n){
  vec[i] = vec[i]*2
}
```

A loop by name looks similar:

```
> vec = rep(1, n)
>
> names(vec) = paste("name", 1:length(vec), sep="")
>
> vec
 name1  name2  name3  name4  name5  name6
     1      1      1      1      1      1


> for (n in names(vec)){
+   vec[n] = vec[n]*2
+ }
> vec
 name1  name2  name3  name4  name5  name6
     2      2      2      2      2      2
```

As you can see, the only visible difference between these two loops is how the element in the vector is referenced. Names are convenient for people, but not as much for the computer. To access an element by name, the computer has to look for it, like in the example with the softball ceremony. The substitute coach had to start from the beginning of the line to find the right person, similarly, the computer has to start from the beginning of the data structure to find an element with the correct name. As a result, the complexity of the loop by name is no longer linear, it's quadratic:

$1 + 2 + 3 + 4 + ... = n(n + 1)/2 = O(n^2)$

The difference in efficiency between the two loops, by index and by name, is demonstrated in the following graph.

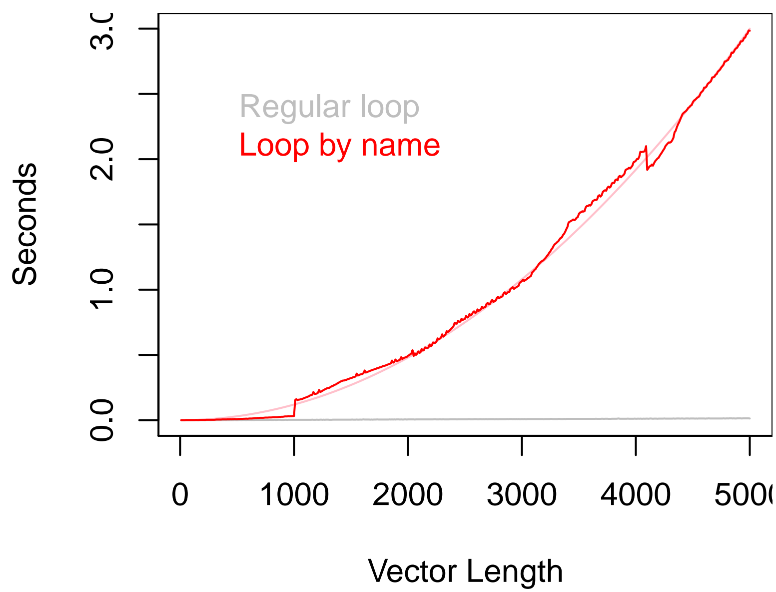


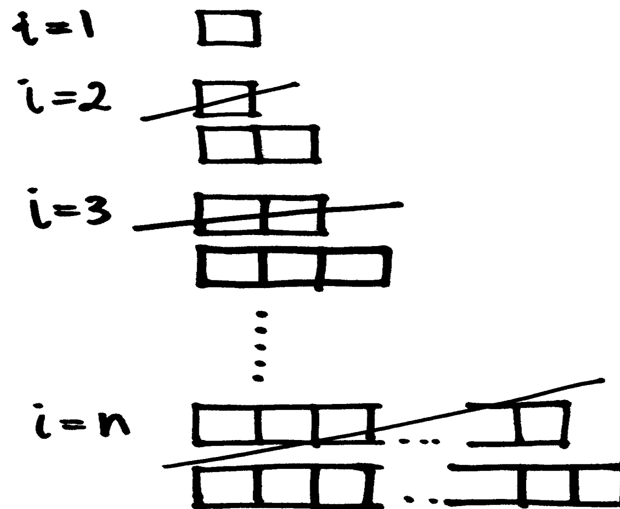
Figure 4: Graphical comparison of efficiency of regular loop and loop by name

6.2 Loops with c(), cbind(), rbind()

More often than loop by name, we use loops with functions c(), cbind(), or rbind(). For example:

```
vec = c()

for (i in 1:n){
  vec = c(vec, 2)
}
```

The code shows that we start with an empty vector and we "grow" this vector inside the loop. During each iteration i of the loop, the computer erases the previous vector of length i-1 and creates a new vector of length i. This process is illustrated by the following diagram:



As seen from the diagram, the computer works harder and harder on each iteration of the loop. Its complexity is $1 + 2 + 3 + 4 + ... = n(n+1)/2 = O(n^2)$. To what degree this loop is less efficient than a loop by index is shown in the following graph:

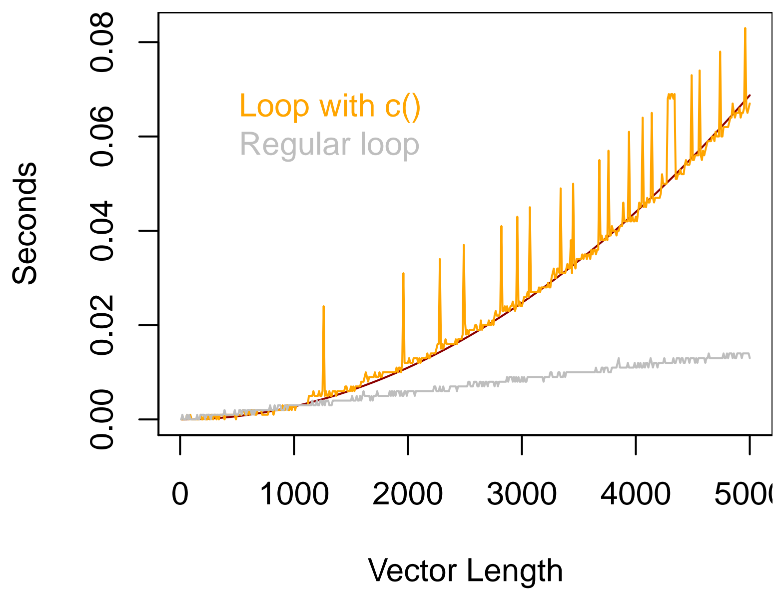


Figure 5: Graphical comparison of efficiencly of regular loop and loop with c()

Let's also see the difference in efficiency between these two loops for several vector lengths:

Vector length	Regular loop	c() loop
10,000	0 sec.	0.3 sec.
100,000	0.3 sec.	28 sec.
1,000,000	2.9 sec.	46 min.
10,000,000	29 sec.	3 days
100,000,000	5 min.	318 days

Similar to c(), using functions cbind() or rbind() inside a loop to "grow" a data structure leads to quadratic complexity and should be avoided.

6.3 Loops and subsets

Suppose we write a loop that iterates on subject ID, and on each iteration it subsets the data frame on the current ID. Such loops will also have quadratic complexity, because when we subset a data frame, either by using

```
subset(data, id==333)
```

or by using

```
data[data$id==333, ]
```

the computer has to search all IDs on each iteration making sure it found every single one, which will result in complexity of $n + n + n + n + ... = n * n = O(n^2)$

7 Summary

- Vector operations are the fastest, but cannot perform all operations.
- Loops by index are not as fast, but can do everything.
- sappy() on vectors is less efficient than loops by index.
- Loops by name can be handy, but should be avoided for large data
- Loops with c(), cbind(), or rbind() should be avoided for large data.
- Loops with subset(data, id==333), or data[data$id==333,] should be avoided for large data.

8 Example

8.1 Single record per ID

Let's see an example of how to improve the efficiency of a loop. Suppose we have a data frame.

```
> data = data.frame(id=c("AAA", "BBB", "CCC"), val=c(112, 56, 99))
> data
  id val
1 AAA 112
2 BBB  56
3 CCC  99
```

We would like to "bootstrap" its records. In other words, we would like to get a random sample with replacement of the data frame records, and this random sample should have the same number of rows as the original data frame. One way of doing this is to sample the IDs, then to subset the data frame on these IDs, and finally to put all sampled data records together.

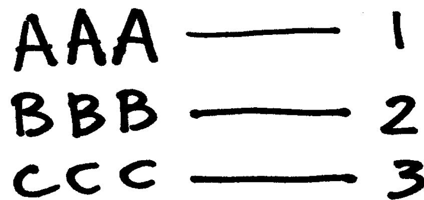
```
> set.seed(1)
> sampledIDs = sample(data$id, size = length(data$id), replace = TRUE)
> sampledIDs
[1] AAA BBB BBB

> bootResult = subset(data, id %in% sampledIDs[1])

> bootResult

   id val
1 AAA 112

> bootResult = rbind(bootResult, subset(data, id %in% sampledIDs[2]))
> bootResult

id val
1 AAA 112
2 BBB  56

> bootResult = rbind(bootResult, subset(data, id %in% sampledIDs[2]))
> bootResult

    id val
1  AAA 112
2  BBB  56
21 BBB  56
```

Using a regular loop we get the following:

```
bootIDs = sample(x=unique(data$id), size=length(uniqueIDs), replace=TRUE)
```

```
bootData = data[data$id==bootIDs[1],]

for (i in 2:length(bootIDs)){
  addID = subset(data, id==bootIDs[i])
  bootData = rbind(bootData, addID)
}
```

This loop will work correctly, but, as we learned earlier, inefficiently because of functions subset() and rbind(). Luckily, there is a more efficient way to do the same task. We can sample the indices of the original data frame and use them to retrieve its records. We can do this because each ID in the data frame has a corresponding index:



After sampling the indices:

```
> set.seed(1)
> sampledIndices = sample(1:nrow(data), size = nrow(data), replace = TRUE)
> sampledIndices
[1] 1 2 2
```

We can retrieve the corresponding records:

```
> data[sampledIndices,]
     id val
1   AAA 112
2   BBB  56
2.1 BBB  56
```

8.2 Multiple records per ID

Now, to make things more challenging, let's consider an example with several records per ID:

```
> data

   id val
1 AAA 112
2 AAA 113
3 BBB  56
4 BBB  57
5 BBB  58
6 CCC  99
```

We would like to generate a boostrap sample of this data with the same number of IDs, and each ID has to have the same records as in the original data frame. Let's try to use our fast algorithm and simply bootstrap by index:

```
> sampledIndices = sample(1:nrow(data), size = nrow(data),
    replace = TRUE)
> sampledIndices
[1] 2 3 4 6 2 6

> data[sampledIndices,]
     id val
2   AAA 113
3   BBB  56
4   BBB  57
6   CCC  99
2.1 AAA 113
6.1 CCC  99
```

This did not work correctly: in the resulting data frame, ID "BBB" has only two records instead of three, and ID "AAA" doesn't have the same records as in the original data frame. This happened because when we simply sample indices we don't take into account that they belong to a certain ID.

The first method works correctly (but inefficiently):

```
bootIDs = sample(x=unique(data$id), size=length(uniqueIDs), replace=TRUE)

bootData = data[data$id==bootIDs[1],]

for (i in 2:length(bootIDs)){
  addID = subset(data, id==bootIDs[i])
  bootData = rbind(bootData, addID)
}

  > bootResult
```

```
    id val
1  AAA 112
2  AAA 113
3  BBB  56
4  BBB  57
5  BBB  58
31 BBB  56
41 BBB  57
51 BBB  58
```

Note, that if the number of records were the same for each ID, we could reshape our data into a "wide" format and bootstrap it by index. In our case the number of records is different for each ID, and instead of reshaping, we would like to apply the idea of sampling indices instead of IDs.

```
AAA ——— 1
AAA ——— 2
BBB ——— 3
BBB ——— 4
BBB ——— 5
CCC ——— 6
```

To achieve this we will create a data structure, in which each ID has a single unique index, and at the same time this data structure will keep all indices for each ID:

```
> ### bootstrap preparation

> indPerID = tapply(1:nrow(data), data$id, function(x){x})

> indPerID
$AAA
[1] 1 2

$BBB
[1] 3 4 5

$CCC
[1] 6
```

Now we can sample the indices of this data structure:

```
> set.seed(1)
> bootIDIndices = sample(x=1:length(indPerID),size=length(indPerID),
      replace=TRUE)
```

```
> bootIDIndices

  [1] 1 2 2
```

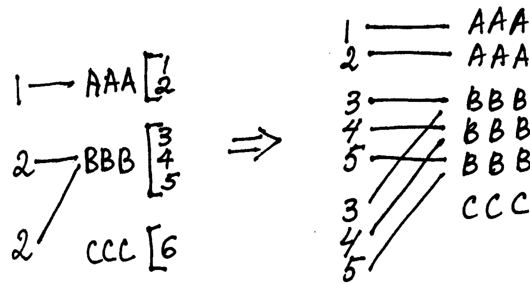
Next we gather the original indices of each ID using the following loop,

```
> newDataLength = sum(lengthPerID[bootIDIndices], na.rm=TRUE)
> lengthPerID = sapply(indPerID, length)
> allBootIndices = rep(NA, newDataLength)
> endInd = 0
> for (i in 1:length(bootIDIndices)){
    startInd = endInd + 1
    endInd = startInd + lengthPerID[bootIDIndices[i]] - 1
    allBootIndices[startInd:endInd] = indPerID[[bootIDIndices[i]]]
    print(allBootIndices)
}

  [i=1]  1  2 NA NA NA NA NA NA

  [i=2]  1  2  3  4  5 NA NA NA

  [i=3]  1  2  3  4  5  3  4  5
```

and retrieve the corresponding rows from the original data frame:

```
> data[allBootIndices, ]

      id val
1    AAA 112
2    AAA 113
3    BBB  56
4    BBB  57
5    BBB  58
3.1 BBB  56
4.1 BBB  57
5.1 BBB  58
```

This idea is illustrated in the following diagram:



Using indices, allows us to improve efficiency of the code

```
bootIDs = sample(x=unique(data$id), size=length(uniqueIDs), replace=TRUE)
bootData = data[data$id==bootIDs[1],]
for (i in 2:length(bootIDs)){
  addID = subset(data, id==bootIDs[i])
  bootData = rbind(bootData, addID)
}
```

by replacing subset() and rbind(), with a loop, where every element of every data structure is referenced by index:

```
bootstrappedIDIndices = sample(x=1:length(indPerID), size=length(indPerID),
          replace=TRUE)
allBootstrappedIndices = rep(NA, newDataLength)
endInd = 0
for (i in 1:length(bootIDIndices)){
  startInd = endInd + 1
  endInd = startInd + lengthPerID[bootIDIndices[i]] - 1
  allBootIndices[startInd:endInd] = indPerID[[bootIDIndices[i]]]
}
```

Was this worth the effort though ? The following table presents processing times for both methods. These times were computed for 200 bootstrap replications and for data frames with random number of records per ID. The codes for both methods can be found in the Appendix.

Num. of IDs	First method	Improved method
100	16 sec.	0.4 sec.
200	3 min.	2.6 sec.
400	49 min.	22 sec.
1000	8.6 hours	2.2 min

9 Aknowledgements.

I would like to thank Cathy Jenkins, Chris Fonnesbeck, Terri Scott, Jonathan Schildcrout, Thomas Dupont, Jeff Blume, and Frank Harrell for their help and useful feedback.

10 Appendix. Example Code.

The following functions generate bootstrap samples by ID. The functions are presented in the order of their efficiency. Note that to use these efficiently in practice, it is better to include your calculations in the function (for example point estimate of the parameter of interest for each bootstrap sample), and return the calculated values.

```
### Least Efficient:
####################
bootstrapByID = function(data, idVar, bootTimes=200){
  data = data[!is.na(data[[idVar]]),]
  uniqueIDs = unique(data[[idVar]])
  for (bi in 1:bootTimes){
    bootstrappedIDs = sample(x=uniqueIDs, size=length(uniqueIDs), replace=TRUE)
    bootstrappedData = data[data[[idVar]]==bootstrappedIDs[1],]
    for (i in 2:length(bootstrappedIDs)){
      bootstrappedData = rbind(bootstrappedData,
                               data[data[[idVar]]==bootstrappedIDs[i],])
    }
  res = bootstrappedData
  }
}
```

```
### More Efficient:
###################
bootstrapByIndexWithUnlist = function(data, idVar, bootTimes=200){
  data = data[!is.na(data[[idVar]]),]
  indPerID = tapply(1:nrow(data), data[[idVar]], function(x){x})
  lengthPerID = sapply(indPerID, length)
  for (bi in 1:bootTimes){
    bootstrappedIDIndices = sample(x=1:length(indPerID), size=length(indPerID),
                                   replace=TRUE)
    res = data[unlist(indPerID[bootstrappedIDIndices]), ]
  }
}
```

```
### Most efficient:
###################
bootstrapByIndex = function(data, idVar, bootTimes=200){
  data = data[!is.na(data[[idVar]]),]
  indPerID = tapply(1:nrow(data), data[[idVar]], function(x){x})
  lengthPerID = sapply(indPerID, length)
  for (bi in 1:bootTimes){
    bootstrappedIDIndices = sample(x=1:length(indPerID), size=length(indPerID),
                                   replace=TRUE)
    newDataLength = sum(lengthPerID[bootstrappedIDIndices], na.rm=TRUE)
    allBootstrappedIndices = rep(NA, newDataLength)
    endInd = 0
    for (i in 1:length(bootstrappedIDIndices)){
      startInd = endInd + 1
      endInd = startInd + lengthPerID[bootstrappedIDIndices[i]] - 1
      allBootstrappedIndices[startInd:endInd] = indPerID[[bootstrappedIDIndices[i]]]
    }
    res = data[allBootstrappedIndices, ]
  }
}
```