Center for Tropical Forest Science R Package Manual
Pamela Hall, Suzanne Lao, Ellen Connell and Marie Massa
Version 1.00 March 29, 2006

7.0 Useful R functions for analysis of CTFS datasets

Synopsis of Contents of this Help File

x %in% table

This is short cut to selecting rows out of a file that have some given value for a variable. It returns a vector of TRUE and FALSE so use it as a conditional.

```
rare.spp<-names(abund.spp$N[abund.spp$N<100])
rare.vct<-bci90.full$sp%in%rare.spp
rare.spp.ind<-bci90.full[rare.vct,]
```

***subset*(x, subset, select)**

Creates a subset of the dataset *x* based a condition *subset*.

```
bci90.bigtree <- subset(bci90.full, bci90.full$dbh>=300)
```

order(vector1,vector2...,decreasing=FALSE)

Provide a vector to be sorted, returns the order number of each value in the svector. Use this number to reorder the initial vector or data frame.

```
hameax[order(hameax$dbh,decreasing=TRUE),][1:10,]
```

sort(vector,decreasing=FALSE)

Provide a vector, returns the sorted vector. This can only be used on a vector.

```
sort(hameax$dbh,decreasing=TRUE)[1:10]
```

rank(vector, ties.method = c("average", "first", "random"))

Provide a vector to be ranked, returns the rank number of each value in the vector. Use with order to create a vector or data frame in rank order.

```
hameax[order(rank(hameax$dbh)),][1:10,]
```

tapply(vector,index.vector,function)

Provide a vector and an index vector of the same length. The index vector contains categories for classification of the values in the first vector. Apply the function to the values in each category. Eg. compute the mean dbh0 for dbh classes.

```
tapply(bci90.full$dbh,dbh.vct,mean)
```

apply(vector,row or col number,function)

Provide a maxtrix, use row or column values to classify the other values into categories. Apply the function to values in the column or row. This computes the number of habitats for which the species has more than 10 trees.

```
abund.spp.ha <- abundance(bci90.full,split1=bci90.full$sp,
     split2=hab.vct)
countspp=function(x,min) return(length(subset(x,x>min)))
```

```
apply(abund.spp.ha,1,countspp,10))
```

merge(x, y, by.x = by, by.y = by, all.x = FALSE, all.y = FALSE)

Provide 2 data frames (x and y) and the columns used to merge the 2 files (by.x, by.y). Indicate whether all of the rows in x and/or all of the rows in y should be preserved. Often only all of the rows in x should be preserved.

```
bci90.full.grform <- merge(bci90.full,bcispp.info[,c(1,5)],
by.x="sp",by.y="sp",all.x=TRUE)
```

match(vector, table, nomatch = NA)

Provide 2 vectors. Returns a vector of the row numbers in table the same length as the first vector when the values in the vectors match.

```
spp.mtch <- match(bci90.full$sp,bcispp.info$sp,nomatch=NA)
table(spp.mtch)
spp.mtch
   10   234   262
11128   160  1133

bcispp.info[c(10,234,262),c(1,5:8)]
            sp grform repsize breedsys maxht
alsebl alsebl       T      20        B    15
psycde psycde       S       1        B     3
socrex socrex       M       8        M    30
```

cut(x,breaks, labels = NULL, right = TRUE)

Creates a categorical value for a continuous variable using the class limits provided by *breaks*.

7.1 Examples and more detailed description of use

x %in% table

This is short cut to selecting rows out of a file that have some given value. It tests a condition and so returns a vector of TRUE and FALSE.

x the values to be matched.
table the values to be matched against.

%in% is actual the function:

"%in%" <-function(x, table) match(x, table, nomatch = 0) > 0

Here is an example of how to use *%in%* which greatly simplifies the process of getting "interesting" populations to analyses.

Problem: Get the tree data for rare species, defined as species with less than 100 individuals.

First, run *totalabund()* for each species. The result is a list and the first object is the abundances for each species.

```
abund.spp<-abundance(bci90.full,split1=bci90.full$sp)
abund.spp$N[1:7]
acacme  acaldi  acalma  ade1tr  aegipa  alchco  alchla
    11     821      44     279      92     266       3
```

Identify the rare species, defined here as species with less than 100 individuals.

```
rare.spp<-names(abund.spp$N[abund.spp$N<100])
rare.spp[1:7]
[1] "acacme" "acalma" "aegipa" "alchla" "amaico" "anacex" "annoha"
```

Use *%**in**%* to create a vector of TRUE and FALSE that is the same length as the full dataset and identifies the rows in the full dataset that are the individuals of rare species. Note the length of *rare.vct* is the same as *bci90.full* and that when the values are tabled, there are 8235 trees that are of rare species.

```
rare.vct<-bci90.full$sp%in%rare.spp
length(rare.vct)
[1] 344846
table(rare.vct)
rare.ind
 FALSE   TRUE
336611   8235
```

Use *rare.vct* as the conditional vector to select the rows from *bci90.full* that are trees of rare species. Note that *rare.ind* is a data frame containing all the columns of the full datasets and is of the same length as the tabled TRUE values.

```
rare.ind<-bci90.full[rare.vct,]
dim(rare.ind)
[1] 8235   14
rare.ind[1:3,1:7]
     tag     sp    gx    gy dbh0 dbh1 pom0
1 105951 acacme 610.0 104.7  105  116    1
2 132160 acacme 534.8 241.3   85   91    1
3 132234 acacme 539.4 242.3  119  122    1
```

Here's a check that the trees records have been properly identified and subsetted. Using the results from *abundance()* the sum of the abundances of all rare species is 1990

is computed. Then the abundance of the new data frame *rare.ind* is computed using *abundance()*. Remember, for any set of tree records, not all of them will be valid as "alive" in a given census, so the number of records in the data frame is NOT the same as the abundance of rare species.

```
sum(abund.spp$N[abund.spp$N<100])
[1] 4589
abundance(rare.ind)$N
$N
      all
all  4589
```

In summary, the steps are as follows. Note that the conditional can be used directly with *%**in**%* instead of making an intermediary vector.

```
abund.spp<-abundance(bci90.full,split1=bci90.full$sp)
rare.spp<-names(abund.spp$N[abund.spp$N<100])
rare.ind<-bci90.full[bci90.full$sp%in%rare.spp,]
```

subset(x, subset, select)

x	object to be subsetted.
subset	logical expression.
select	expression, indicating columns to select from a data frame.

subset() makes a subset of the data in *x* based on the condtion provided by *select*. In addition *subset()* creates a new object that is just those subsetted records and removes all records that have NA for the conditions of *select*. The *select* argument accepts a conditional statement and turns it into a *vector* of TRUE and FALSE or it accepts the name of a variable in the data object *x* and some condition of it. You don't have to respecify the objects' name only the variable within.. Recall that the use of the *[condition]* includes all NAs and does not directly create a new data object, it only creates a vector of TRUE or FALSE which can be used for subsetting. Here are some examples.

```
bci90.bigtree <- subset(bci90.full,bci90.full$dbh>=300)
```

```
> bci90.bigtree[1:5,1:6]
      tag     sp    gx    gy dbh pom
2453 3193 ade1tr 639.3 309.4 525   1
2999  207 alchco 972.8 130.3 353   1
3000  256 alchco 967.5 303.4 310   1
3001  279 alchco 973.6 392.9 549   2
3003  834 alchco 889.5   8.1 565   1
```

Here is the same example, but only 3 variables from the full dataset are preserved in the subsetting.

```
bci90.bigtree=subset(bci90.full,bci90.full$dbh0>300,
     select=c(sp,dbh0,dbh1))
> bci90.bigtree[1:5,]
       sp dbh0 dbh1
2  ABARJU  333  335
3  ABARJU  444  472
7  ABARJU  361  378
12 ABARJU  673   NA
14 ABARJU  367  386
```

A more complicated use that uses the function *%**in**%* as part of the *select* condition.

```
abund.spp<-abundance(bci90.full,split1=bci90.full$sp)
abund.spp$N[1:7]

ABARJU ABARLA ABUTGR ACALCU ACALPU
    23     96    293    538      0
```

Identify the rare species, defined here as species with less than 100 individuals.

```
rare.spp<-names(abund.spp$N[abund.spp$N<100])
rare.spp[1:7]
[1] "acacme" "acalma" "aegipa" "alchla" "amaico" "anacex" "annoha"

bci.rare<-subset(bci90.full,bci90.full$sp%in%rare.spp)
> bci.rare[1:5,1:6]
     tag     sp    gx    gy dbh pom
1 105951 acacme 610.0 104.7 105   1
2 132160 acacme 534.8 241.3  85   1
3 132234 acacme 539.4 242.3 119   1
4 132235 acacme 538.8 242.5  29   2
5 191542 acacme 282.7 177.5  41   1
```

order(vector1,vector2...,na.last=TRUE,decreasing=FALSE)

vector1, vector2...
: 1 or more vectors of the same length

na.last
: place NAs at the end

decreasing = FALSE
reorders from low to high or TRUE for high to low. If using more than 1 vector and want one to increase and one to decrease, use a "-" (negative) sign in front of the variable name

order() returns a new vector of row numbers. The row numbers refer to the rows in the original vector in increasing (or decreasing) order. To get the original vector reordered, use the row numbers.

Examples:

```
ocotob$dbh[1:10]
 [1]  NA  NA 350  NA  NA  NA  NA 335  NA  NA

default.order <- order(ocotob$dbh,decreasing=TRUE)
default.order[1:10]
 [1]  14  11 157  15   3   8  20  18  64  95
```

row 14 has rank 1
row 11 has rank 2
row 157 has rank 3...

```
ocotob$dbh[c(14,11,157,15,3,8,20,18,64,95)]
 [1] 444 428 428 405 350 335 324 323 300 282
```

N ote below how the row have been reordered according to the value of *default.order*

```
ocotob[default.order,][1:4,1:7]
          tag      sp    gx    gy dbh pom date
225196   3910 OCOTOB 556.5 174.4 444   1 3514
225193   2513 OCOTOB 716.8 221.4 428   1 3570
225339 209296 OCOTOB 221.2 491.8 428   1 3506
225197   4005 OCOTOB 529.5 148.8 405   2 3674
```

Put it all together in one line:

```
ocotob[order(ocotob$dbh,decreasing=TRUE),][1:4,1:7]
          tag      sp    gx    gy dbh pom date
225196   3910 OCOTOB 556.5 174.4 444   1 3514
225193   2513 OCOTOB 716.8 221.4 428   1 3570
225339 209296 OCOTOB 221.2 491.8 428   1 3506
225197   4005 OCOTOB 529.5 148.8 405   2 3674
```

Use of more than 1 vector:

```
hameax <- bci90.split$HAMEAX
hameax[1:5,1:7]
          tag      sp    gx    gy dbh pom date
```

```
127020 15377 HAMEAX 845.3  59.2  NA  NA 3557
127021 15984 HAMEAX 841.7 138.0  16   1 3572
127022 17883 HAMEAX 858.7 266.1  NA   2 3592
127023 19466 HAMEAX 852.3 475.9  NA  NA 3612
127024 22527 HAMEAX 947.5 313.6  NA  NA 3639

tmp<-order(hameax$dbh,decreasing=FALSE)
tmp[1:5]
[1] 115 138 202 139 114

hameax[tmp,][1:5,1:7]
          tag     sp  gx    gy dbh pom date
127134 265824 HAMEAX 2.8 497.2  11   1 3431
127157 400428 HAMEAX 6.4 352.0  10   1 3397
127221 500193 HAMEAX 7.5 351.2  NA  NA 3397
127158 400500 HAMEAX 7.9 375.1  12   1 3403
127133 261744 HAMEAX 8.4 200.1  NA  NA 3366

tmp<-order(hameax$gx,hameax$gy,decreasing=FALSE)
tmp[1:5]
[1] 115 138 202 139 114

hameax[tmp,][1:5,1:7]
          tag     sp  gx    gy dbh pom date
127134 265824 HAMEAX 2.8 497.2  11   1 3431
127157 400428 HAMEAX 6.4 352.0  10   1 3397
127221 500193 HAMEAX 7.5 351.2  NA  NA 3397
127158 400500 HAMEAX 7.9 375.1  12   1 3403
127133 261744 HAMEAX 8.4 200.1  NA  NA 3366
```

Use of "-" to change the order from increasing to decreasing. Note that only the reordered row numbers are provided below, not the actual data of the species. This is particularly useful when using 2 vectors but you want the sort order opposite in them: eg. the first decreasing and the second increasing.

```
tmp<-order(hameax$dbh,hameax$date)
tmp[1:5]
[1] 152 153 155 138 160
tmp<-order(hameax$dbh,-hameax$date)
tmp[1:5]
[1] 165 160 138 153 155
```

sort(vector,na.last=TRUE,decreasing=FALSE)

vector
: only one vector can be used

na.last
: place NAs at the end

decreasing = FALSE
: reorders from low to high or TRUE for high to low.

sort() returns the actual vector in a different order, not a set of row numbers as in *order()*. Therefore, the results of *sort()* cannot be used to reorder a data frame.

Examples:

```
hameax[1:6,1:7]
          tag     sp    gx    gy dbh0 dbh1 pom0
127020 15377 HAMEAX 845.3  59.2   NA   NA 3557
127021 15984 HAMEAX 841.7 138.0   16    1 3572
127022 17883 HAMEAX 858.7 266.1   NA    2 3592
127023 19466 HAMEAX 852.3 475.9   NA   NA 3612
127024 22527 HAMEAX 947.5 313.6   NA   NA 3639
127025 24557 HAMEAX 957.2 497.9   NA   NA 3639

sort(hameax$dbh,decreasing=FALSE)[1:6]
[1] 10 10 10 10 10 10
```

Beware of using "-" in *sort()*. It does NOT behave the same as in *order()*. The negative sign negates the values, it does not change the sorting order.

```
sort(hameax$dbh)[1:6]
[1] 10 10 10 10 10 10
sort(hameax$dbh,decreasing=TRUE)[1:6]
[1] 39 37 34 31 31 30
sort(-hameax$dbh)[1:6]
[1] -39 -37 -34 -31 -31 -30
```

rank(vector, na.last = TRUE,
ties.method = c("average", "first", "random"))

vector
: only one vector can be used

na.last
: place NAs at the end

ties.method
: provides options for determining how to assign a rank to tied values. Use of *average* will return a decimal values.

rank() returns the rank value of the vector contents in the order of the original vector. You can combine *rank()* and *order()* to create a data frame sorted by the rank order of a given column within the data frame. Note how *ties.method=("random")* creates different

order for rows with equal ranked values.

Examples:

```
hameax[1:5,1:7]
          tag     sp     gx     gy dbh dbh1 pom0
127020 15377 HAMEAX 845.3   59.2   NA   NA 3557
127021 15984 HAMEAX 841.7 138.0   16    1 3572
127022 17883 HAMEAX 858.7 266.1   NA    2 3592
127023 19466 HAMEAX 852.3 475.9   NA   NA 3612
127024 22527 HAMEAX 947.5 313.6   NA   NA 3639

tmp <- rank(hameax$dbh)
tmp[1:7]
 [1] 108  55 109 110 111 112 113
hameax[rank(hameax$dbh),][1:5,1:7]
          tag     sp     gx     gy dbh pom date
127127 242878 HAMEAX   88.7 280.9   NA   NA 3368
127074 116524 HAMEAX 564.8 180.5   NA   NA 3502
127128 242917 HAMEAX   91.0 294.6   NA   NA 3368
127129 244848 HAMEAX   84.8 486.7   NA   NA 3407
127130 246496 HAMEAX   68.5 171.2   NA   NA 3347

tmp <- order(rank(hameax$dbh,ties.method="random"))
tmp[1:7]
 [[1] 153 152 155 160 165 138 161

hameax[order(rank(hameax$dbh,ties.method=c("random"))),][1:5,1:7]
          tag     sp     gx     gy dbh0 dbh1 pom0
127157 400428 HAMEAX    6.4 352.0   10    1 3397
127171 410049 HAMEAX 207.9   36.5   10    1 3333
127179 413436 HAMEAX 265.2 245.7   10    1 3431
127172 410373 HAMEAX 213.6 229.7   10    1 3361
127174 410406 HAMEAX 218.4 230.0   10    1 3361
```

tapply(vector, index, FUN = NULL)

vector
: one vector only

index
: a *vector* or *list* of the classes into which the values in the input *vector* are going to be categorized. The *index* must be of the same length as the input *vector*. It can be numeric or character. More than one *index vector* can be used. It must be provided as a *list.*

FUN

the function to be applied to the *vector* values in each class of *index*. In the case of functions like +, %*%, etc., the function name must be quoted.

tapply() works as a loop. It categorizes each value in *vector* by the classes in *index* treating these as a "population". Then it performs the *FUN* on the population. For example, *vector* = growth, *index* = dbhcat, *FUN* = mean, gives the mean growth rate for each dbhclass in dbhcat: *tapply(growth,dbhcat,mean)*

Examples: dbhcat = dbh classes, habcat = habitat designations for quadrates

```
dbhcat <- sep.dbh(bci90.full)
table(dbhcat)
dbhcat
   10.100   100.300 300.10000
   222826     17113      4120
tapply(bci90.full$dbh,dbhcat,mean)
   10.100   100.300 300.10000
 28.21493 157.41033 503.21917
habcat <- sep.quadinfo(bci90.full,bciquad.info,by.col="hab")
table(habcat)
habcat
    1     2     3     4     5     6     7     8
47250 23650 77806 27770 46752 56487 35292 29752

table(habcat,dbhcat)
       dbhcat
habcat  10.100 100.300 300.10000
     1  30779    2208        574
     2  15234    1091        280
     3  49801    4130        909
     4  18178    1416        327
     5  30698    2403        568
     6  36092    2719        682
     7  22957    1706        419
     8  19044    1440        361

tapply(bci90.full$dbh,list(habcat,dbhcat),mean)
     10.100   100.300 300.10000
1 27.96319 157.9524   516.0540
2 27.61461 155.5481   531.1286
3 28.82796 159.2973   485.6898
4 28.39889 156.7394   500.2966
5 28.08665 157.0300   506.0528
6 27.60972 157.5542   498.6496
```

```
7 28.45973 156.9015   490.8282
8 28.41273 154.2035   526.5069
```

apply(array, dim number, function)

array
: matrix of 2 dimensions or an array of >2 dimensions

number
: the dimension number of categories used to classify the value of the other matrix dimension

function
: the function to be applied to the *matrix* values in each class of in the dimension *number*. In the case of functions like +, %*%, etc., the function name must be quoted.

apply() works as a loop. It starts with the *array* and computes *function* for the values in one dimension by the categories in the second dimension. The dimensions are 1 for row, 2 for column in a 2 dimensional array (a matrix). It is like *tapply()* but the values for the categories and the variable are both contains in the supplies array.

Examples: species and hectares, compute the number of species per hectare and the number of hectares in which a species occurs. The function *countnonzero()* is used instead of length because length will include hectares without species.

```
hacat<-gxgy.to.index(bci90.full$gx,bci90.full$gy,gridsize=100)
sppcat<-bci90.full$sp
abund.spp.ha<-table(sppcat,hacat)
abund.spp.ha[1:5,1:5]
        hacat
sppcat    1  2  3  4  5
  acacme  1  0  0  0  0
  acaldi 64 23  6 20 43
  acalma 19  4  4 13  5
  ade1tr  0  1 33 17  3
  aegipa  8  4  2  7  5
```

Count the species in each ha

```
spp.ha<-apply(abund.spp.ha,2,countnonzero)
spp.ha[1:7]
   1   2   3   4   5   6   7
 208 189 192 192 194 196 188
```

Count the number of ha each species occurs in

```
nspp.ha <- apply(abund.spp.ha,1,countspp,0)
nspp.ha[1:7]
acacme acaldi acalma ade1tr aegipa alchco alchla
```

```
     6     50     23     39     48     49      3
```

apply() can also work on a 3 or more dimension array. The number of dimensions specified provide the dimensions over which the values are aggregated for computing some function. Here is an example with abundance computed for each species and hectare and dbhclass.

```
abund.spp.ha.dbh <- table(sppcat,hacat,dbhcat)
dim(abund.spp.ha.dbh)
[1] 318  50   3
```

Use *apply()* with for 2 dimension to count for that dimension.

```
test1<-apply(abund.spp.ha.dbh,c(1,2),countspp,0)
test1[1:5,1:5]
        hacat
sppcat   1 2 3 4 5
  acacme 1 0 0 0 0
  acaldi 1 1 1 1 1
  acalma 1 1 1 1 0
  ade1tr 0 1 2 2 1
  aegipa 1 1 1 1 2

test1<-apply(abund.spp.ha.dbh,c(1,3),countspp,0)
test1[1:5,]
        dbhcat
sppcat    10.100 100.300 300.10000
  acacme      6       2         0
  acaldi     50       2         0
  acalma     16       2         0
  ade1tr     33      27         1
  aegipa     36      17         0

test1<-apply(abund.spp.ha.dbh,c(2,3),countspp,0)
test1[1:5,]
     dbhcat
hacat 10.100 100.300 300.10000
    1    178      86        37
    2    166      80        40
    3    172      82        41
    4    165      88        22
    5    163      89        38
```

merge(x, y, by = intersect(names(x), names(y)), by.x = by, by.y = by,

all = FALSE, all.x = all, all.y = all)

x, y
: data frames.

by.x, by.y
: specifications of the common columns. This can be the column numbers or names (in quotes).

all.x
: logical; if TRUE, then extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have NAs in those columns that are usually filled with values from y. The default is FALSE, so that only rows with data from both x and y are included in the output.

all.y
: logical; analogous to all.x above.

all
: logical; all=L is shorthand for all.x=L and all.y=L. Note that the default is to EXCLUDE rows that have no match in either data frame.

This function is similar to "lookup" functions in other programs. It take the value of a column in the first data frame (*by.x*) and "looks it up" in the second data frame. If it finds a matching variable value (*by.y*), it merges the columns at that row in the second data frame with those in the first data frame. It merges ALL of the columns, so if you only want 1 (or a few) from the second data frame to be on the first, specify only those that you want to merge and the variable that matches the two data frames together. Take care with the default for *all* that can result in the loss of records from the first data frame if there is no match in the second.

This example merges the growth form from *bci.spp.info* to *bci9095.full*.

```
tst.merge<-merge(bci90.full,bcispp.info,by.x="sp",by.y="sp")
> names(tst.merge)
[1] "sp"   "tag" "gx"   "gy"   "dbh"   "pom"   "date"
[8] "codes" "status" "genus" "species" "family" "grform""repsize"
[15] "breedsys" "maxht"
```

Select only the column(s) in bci.spp.info that is to be merged.

```
tst.merge<merge(bci90.full[,c(1,2,5)],bci.spp.info[,c(1,5)],
     by.x="sp",by.y="sp")
names(tst.merge)
 [1] "sp"   "tag"    "dbh" "grform"
```

match(vector, table, nomatch = NA)

vector

table
Provide 2 vectors: vector and table. Returns a vector of the row numbers in table the same length as the first vector when the values in the vectors match.
nomatch=NA
Use NA when a match cannot be made.

Match returns a vector of row numbers that can be used to select the column values from the second vector that match the values in the first. For example, to find the row in *table.in* with the same species as a row in *datafile*. Then use this vector of row numbers to select rows from *table.in* to get *grform* and create a vector of the same length and order as *datafile.*

This example creates a vector of growth form for each tree in a full dataset.

```
bcispp.info[1:5,c(1,5:8)]
       sp grform repsize breedsys maxht
1 acacme      U       4        B     6
2 acaldi      S       2        M     6
3 acalma      U       2        M     5
4 ade1tr      U      10        D     5
5 aegipa      M       4        B    15

spp.match=match(bci90.full$sp,bcispp.info$sp,nomatch=NA)
spp.match[10:20]
[1] 1 1 1 2 2 2 2 2 2 2 2

spp.grform<-bcispp.info[spp.match,"grform"]
spp.grform[10:20]
[1] "U" "U" "U" "S" "S" "S" "S" "S" "S" "S" "S"
```

cut(x,breaks, labels = NULL, right = TRUE)

x
A vector of the continuous variable that is going to be used to make categories
breaks
The min and max of each category.
labels = NULL
Labels for each category. Default labels are provided (see below).
right = TRUE
How to determine whether the values in a category are >= or only > to the minimum value. For instance, using dbh classes:
right = TRUE 10 >= 100 , 10 is NOT included and 100 IS
right = FALSE 10 => 100 , 10 IS included and 100 is NOT
The CTFS convention is, *right = FALSE*:
10 <= tree dbh < 100

```
bci90.full$dbh[1:10]
 [1] 105  85 119  29  41  18  20  13  18  12>

dbhclass=c(10,30,50,100,300,700,100000)
bci90.dbh.vct  <-  cut(bci90.full$dbh,breaks=dbhclass,right=F)

bci90.dbh.vct[1:7]
[1] [100,300) [50,100)  [100,300) [10,30)   [30,50)   [10,30)
[10,30)
Levels: [10,30) [30,50) [50,100) [100,300) [300,700) [700,1e+05)

bci90.dbh.vct  <-  cut(bci90.full$dbh,breaks=dbhclass,
     right=F,labels=paste(dbhclass[-7],dbhclass[-1],sep="."))
bci90.dbh.vct[1:7]
[1] 100.300 50.100  100.300 10.30   30.50   10.30   10.30
Levels: 10.30 30.50 50.100 100.300 300.700 700.1e+05
```

7.2 How NA , NaN and Inf are handled by some R functions:

NA and and NaN (not a number) are special characters in R. They do not appear quoted. They are either values in an object explicitly used by the design of the object or they are the result of a computation. NaN will be the result a numerical computation that produces an impossible value such as taking the log of a negative number.

```
log(-1)
[1] NaN
Warning message:
NaNs produced in: log(x)
```

Normal arithmetic returns NA if or NaN, respectively, if they are in the computation.

```
x=c(seq(1:5),NA,seq(6:10))
x
 [1]  1  2  3  4  5 NA  1  2  3  4  5
y=c(seq(1:10),NA)
y
 [1]  1  2  3  4  5  6  7  8  9 10 NA
z<-x+y
z
 [1]  2  4  6  8 10 NA  8 10 12 14 NA
```

Inf is a number. It is treated as a number. Therefore, any computation will return Inf or -Inf as found.

```
x
 [1]  1  2  3  4  5 NA  1  2  3  4  5
y
 [1]  1  2  3  4  5  6  7  8  9 10 Inf
z<-x+y
z
 [1]  2  4  6  8 10 NA  8 10 12 14 Inf
```

Inf is also the result of a computation that produces infinity such as dividing by 0.

```
10/0
[1] Inf
```

To test for the presence of NA use:

is.na(x)

This is a condition test. If *x* is an NA the result is FALSE. If *x* is a vector then the result will be FALSE wherever NA is in the vector and TRUE where it is not. A common syntax to get rid of NAs is to negate the test with a !:

!is.na(x)

The descriptive statistics funtions require explicit removal of NA in order to provide meaningfull results. They will return NA if NAs are in the vector of values to be computed. You MUST explicitly remove NA to get a meaningfull answer. To be sure the functions work well, use the full word TRUE not just T.

mean
sum
range
min
max
median
sd

```
mean(bci90.full$dbh)
[1] NA
mean(bci90.full$dbh,na.rm=TRUE)
[1] 45.29252
```

subset() and *[condition]* treat NAs differently. *subset()* will set NAs to FALSE and does NOT INCLUDE them in the subsetted data if they are encountered during the determining of the condition part of subset. Conditions determined by the use of *[ ]* will set NAs to TRUE and DOES INCLUDE them in any subsequent use of the condition vector created.

table() by defaults EXCLUDES NAs. To include them you must explicitly request it:

```
table(bci90.full$dbh)[c(1:5,1021)]
10    11    12    13    14  <NA>
 8301 12851 12597 11474 10882    NA

table(bci90.full$dbh,exclude=NULL)[c(1:5,1021)]
10     11     12     13     14   <NA>
  8301  12851  12597  11474  10882 100787
```

cut() ignores NAs and does not include them in any category.

Functions such as *lm()* will fail if NaN or Inf are included in the data for the model.

```
lm(y~x)
Error in lm.fit(x, y, offset = offset, singular.ok = singular.ok,
```

```
...) :
NA/NaN/Inf in foreign function call (arg 4)
```