Contents

1 The "carry forward" problem

An example of the "carry forward" problem															
seq_along	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
values	89.7	26.6	37.2	57.3	90.8	20.2	89.8	94.5	66.1	62.9	6.2	20.6	17.7	68.7	38.4
value > 50	T	F	F	T	T	F	T	T	T	T	F	F	F	T	F

In the situarion depicted above, we sometimes wish to replace values for which a condition (in the example shown above, "value > 50") is `FALSE` with the most recent value for which that same condition was `TRUE`, moving from left to right. This is the "carry forward" problem. For example, we might want to fill **NA** values with the last valid observation in a time-series of measurments. Because this problem occurs frequently, we need a fast formula for this. R loops would be way too slow.

- We can compute the result directly using the `rep` function with the vector version of the `times` argument
- We can compute indices into the original vector that produce the desired result using `cummax` on the "squelched" `seq_along` vector

1.1 The solution using `rep`

The `rep` function with a vector `times` argument allows us to replicate individual elements by arbitrary counts. If sum of `times` is equal to the length of the original vector, we will get a result with the same length as the original.

Consider the vector 1:10. We might want to replace even values with their preceding odd values.

```
x<-1:10
```

odd values are:

```
x[x%%2==1]

## [1] 1 3 5 7 9
```

and the number of items between odd values is:

```
diff(c(which(x%%2==1),length(x)+1))

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

replicating each element in the "odd" subset by the corresponding element in the repetition vector, gives the desired result:

```
rep(x[x%%2==1],times=diff(c(which(x%%2==1),length(x)+1)))

##  [1] 1 1 3 3 5 5 7 7 9 9
```

1.2 The solution using `cummax`

`cummax` naturally carries forward the recent maximum. We can apply a "mask" vector containing 0 or 1 values to the `seq_along` vector to get the "squelched" `seq_along` vector. Applying `cummax` the "squelched" `seq_along` and obtain a vector of indices
A `seq_along` vector would look like this:

```
1:10

##  [1]  1  2  3  4  5  6  7  8  9 10
```

The "mask" vector looks like this:

```
rep(c(1,0,1),times=c(4,2,4))

##  [1] 1 1 1 1 0 0 1 1 1 1
```

The "squelched" `seq_along` vector will then be:

```
1:10*rep(c(1,0,1),times=c(4,2,4))
```

```
##  [1]  1  2  3  4  0  0  7  8  9 10
```

Applying `cummax` to the "squelched" `seq_along` vector carries forward non-zero values:

```
cummax(1:10*rep(c(1,0,1),times=c(4,1,5)))
```

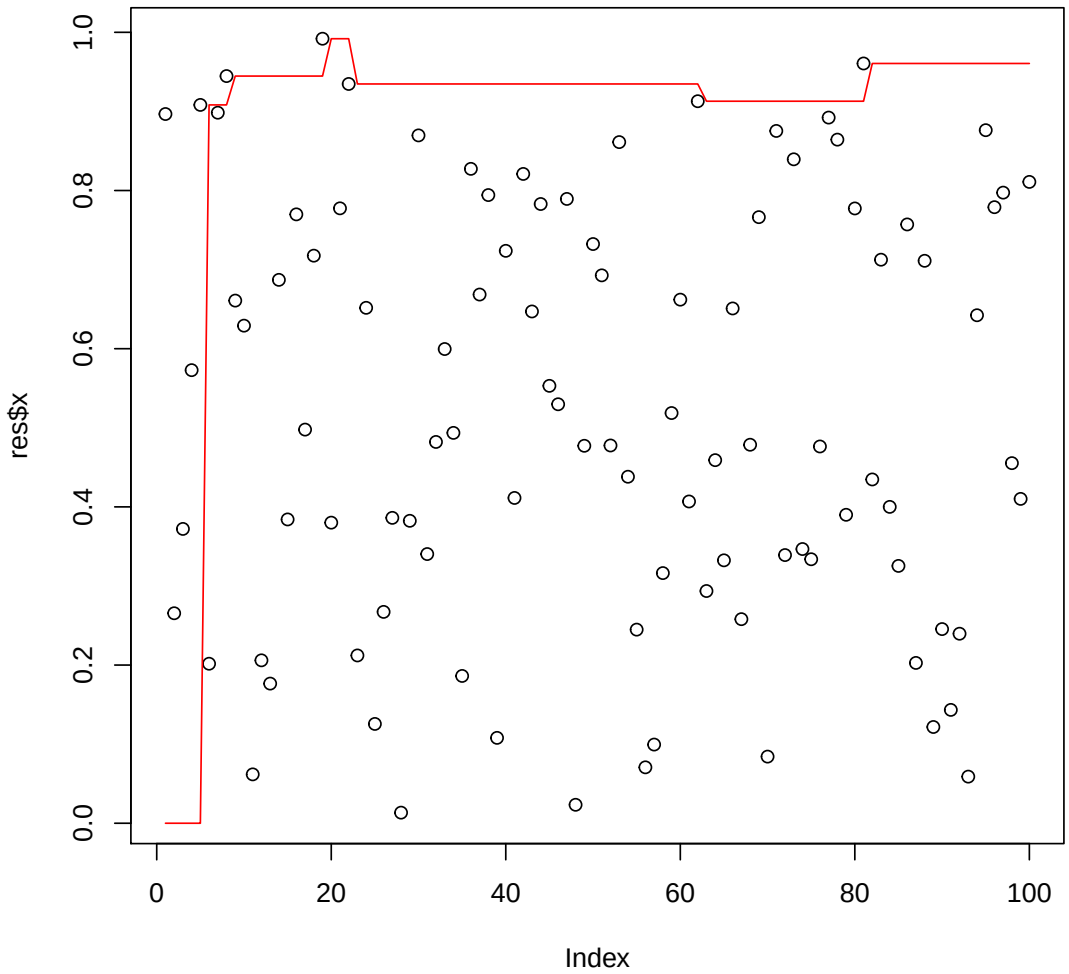
```
##  [1]  1  2  3  4  4  6  7  8  9 10
```

1.3 Carrying forward a value, version 1

We sometimes need to "carry forward" a value at a specific points in a vector. For example, we might want to carry forward non-zero values in a time-series. We can use the `times` argument of the `rep` function to assemble the result

```
set.seed(0)
x<-runif(100)
loc<-x>0.9
loc_diff<-diff(c(0,which(loc),length(x)))
x_cf<-rep(c(0,x[loc]),times=loc_diff)

res<-data.table(
 x=x,
 carry_cforward=x_cf
)
```



1.4 Carrying forward a value, version 2

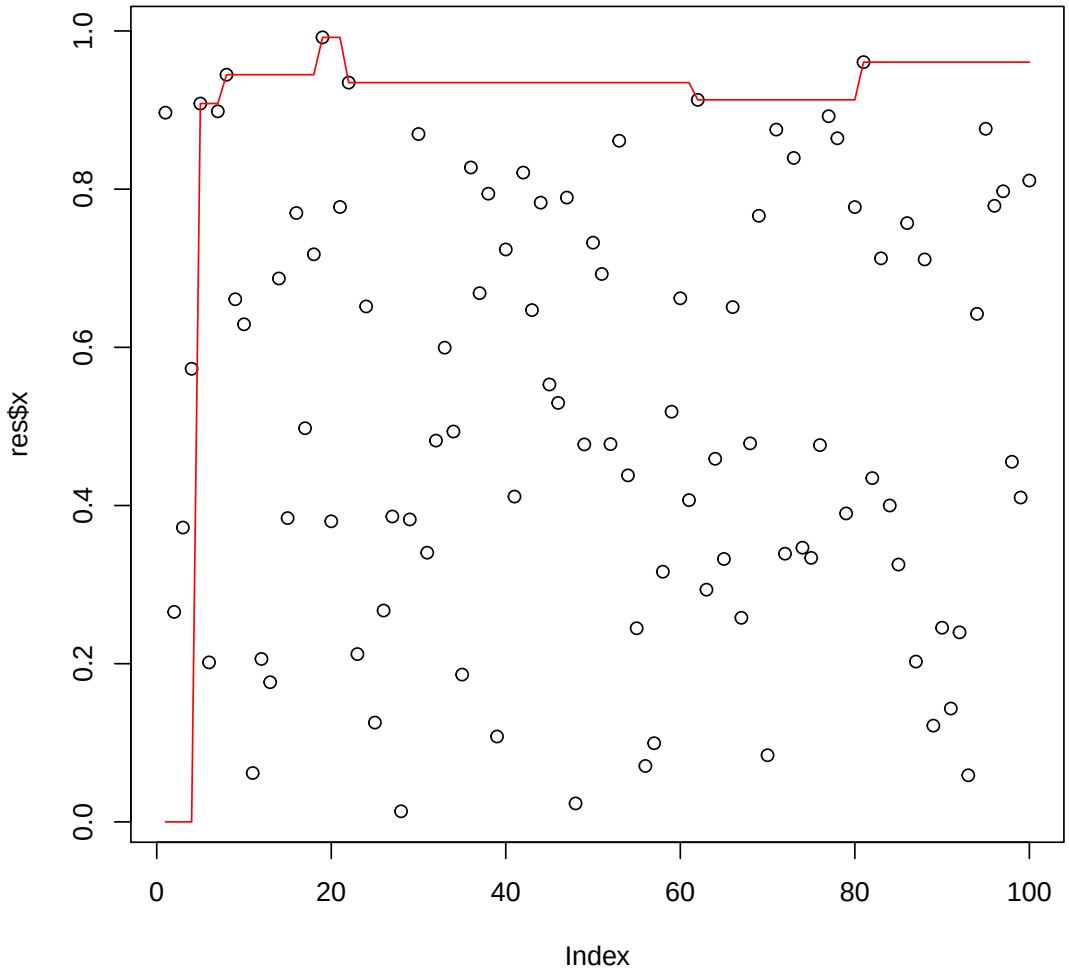
We sometimes need to "carry forward" a value at a specific points in a vector. For example, we might want to carry forward non-zero values in a time-series. We can apply the condition to the `seq_along` vector to produce a vector that has is non-zero and equal to the index of the "carry-forward" values. Because the `seq_along` vector is increasing, we can use `cummax` to do the "carry-forward" operation for us.

```
set.seed(0)
x<-runif(100)
loc<-x>0.9

ndx<-cummax(seq_along(x)*loc)

x_cf<-c(0,x)[ndx+1]

res<-data.table(
 x=x,
 carry_cforward=x_cf
)
```



2 The re-setting count problem

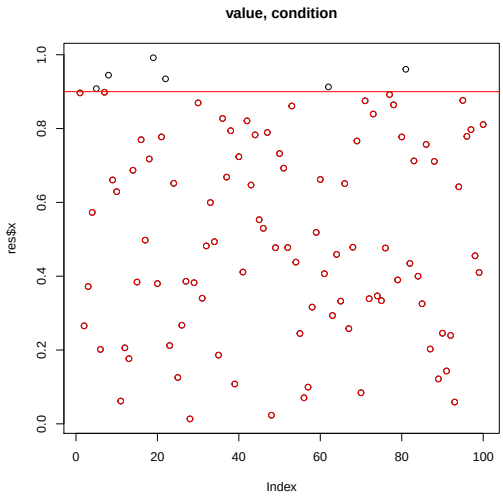
We sometimes need to count the length over which a condition holds. This is called the **re-setting sum**. For example, in the context of some systematic strategies we would like to know how many days ago a signal was triggered. Because of frequent use, a vectorized calculation of the **re-setting sum** is needed.

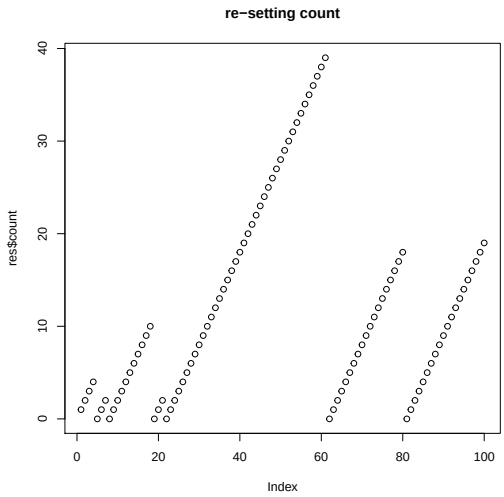
2.1 Re-setting count, solution using `cummax`

```
set.seed(0)
 x<-runif(100)
 test<-x<0.9
 y<-cumsum(test)
 c(0,y-cummax(y*!test))

 res<-data.table(
   x=x,
   test=test*1,
   count=cumsum(test)-cummax(cumsum(test)*!test)
 )
```

re-setting sum of elements since condition is met





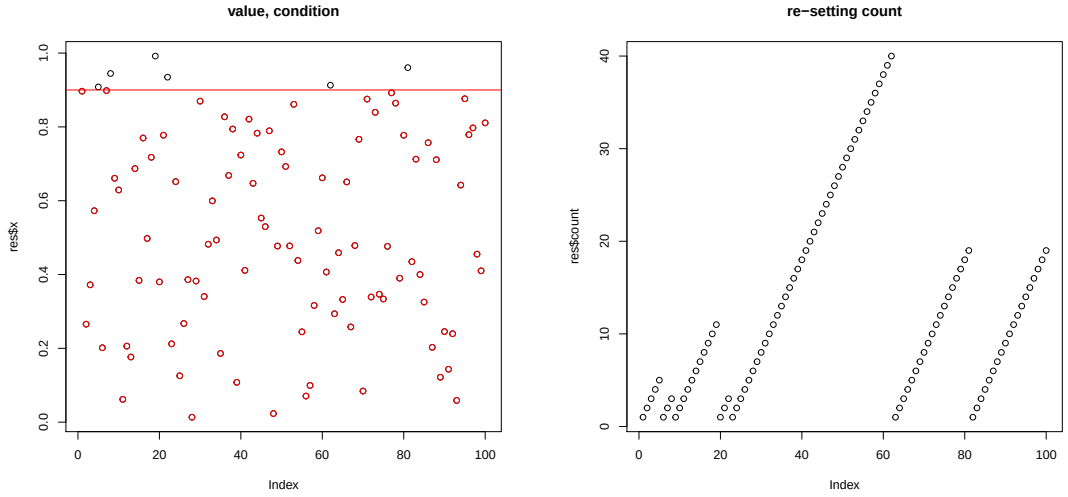
2.2 Resetting count, solution using partial `seq_along` concatenation

We sometimes need to count the length over which a condition holds. This is called the **re-setting sum**. For example, in the context of some systematic strategies we would like to know how many days ago a signal was triggered. Because of frequent use, a vectorized calculation of the **re-setting sum** is needed. We can compute the re-setting sum by explicitly combining subsets of the `seq_along` vector as follows:

```
set.seed(0)
x<-runif(100)
test<-x<0.9
condition_length<-diff(c(0,which(!test),length(x)))
vcs<-mapply(function(i,s)s[1:i],i=condition_length,MoreArgs=list(s=seq_along(x)),SIMPLIFY = FALSE)
count<-do.call(c,vcs)

res<-data.table(
    x=x,
    test=test*1,
    count=count
)
```

re-setting sum of elements since condition is met



3 The outer produce problem

We sometimes need to create all combinations of values in 2 vectors. For example, when creating a grid of strategy performance for all values of two parameters. The `expand.grid` function can be used for this. Some times we want to do this in-line and and avoid the extra function call. The `rep` function has two parameters, `times` and `each` which can be used to achieve this:

```
item1<-paste0("square",1:3)
item2<-paste0("circle",1:3)

combinations<-data.table(
  x=rep(item1,times=length(item2)),
  y=rep(item2,each=length(item1))
)
```

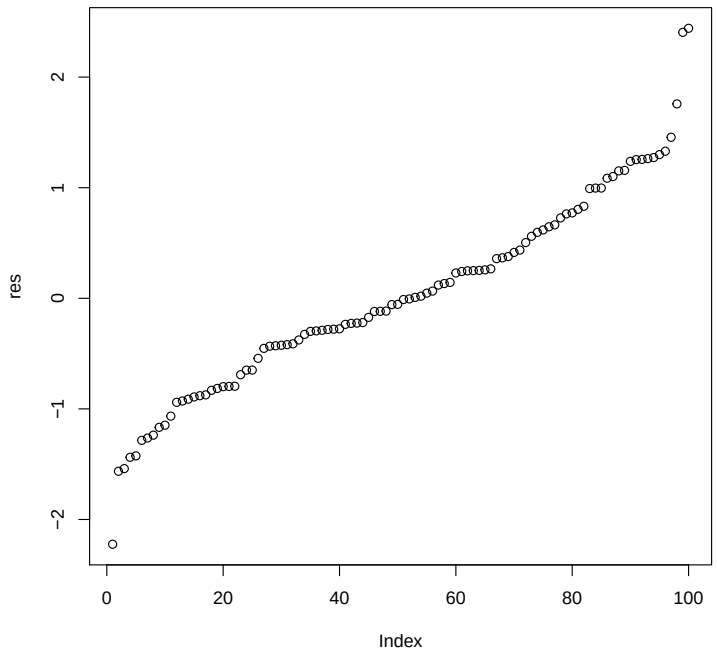
combinations table:

	x	y
1	square1	circle1
2	square2	circle1
3	square3	circle1
4	square1	circle2
5	square2	circle2
6	square3	circle2
7	square1	circle3
8	square2	circle3
9	square3	circle3

4 The sorting permutation matrix problem

It is sometimes useful to express sorting operations in the form of permutation matrices. The reason for this is that we can chain permutations by multiplying the permutation matrices. Dense element ranks are used as column indices into the diagonal matrix to form the permutation matrix.

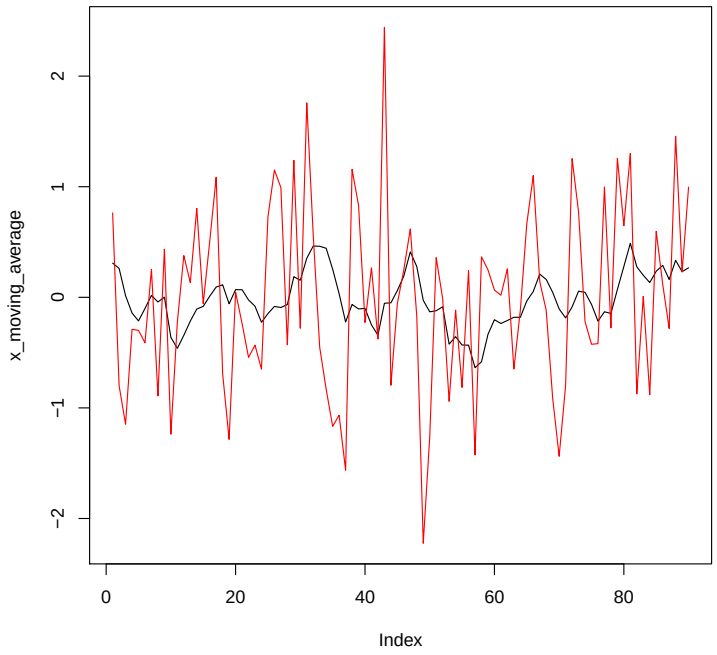
```
set.seed(0)
x<-rnorm(100)
m<-diag(length(x))[,frank(x,ties.method = "dense")]
res<-drop(m%*%cbind(x))
```



5 The moving average problem

Moving averages are basic ingredients for signal calculations. We need a fast method for computing these.

```
set.seed(0)
x<-rnorm(100,mean=0)
x_sum<-cumsum(x)
x_moving_average<-(tail(x_sum,-10)-head(x_sum,-10))/10
```



6 The period stop-loss problem

7 The drawdown statistics problem

Drowdown analysis is an importan component of strategy design. A fast calculation of common statistics is therefore needed. Drawdown statistics are computed daily, so each stat is a vector of values corresponding to a vector of dates, starting at inception and ending at the current date. A strategy's history can be seen as being split by high-watermark dates into a series of **drawdown episodes**. Every day will belong to a single drawdown episode and episodes ordered by episode start date will cover the whole history without gaps. A strategy that trends upwards without downdays will be composed of adjacent 1-day episodes each having a drawdown of zero.

The most common statistics are:

Common drawdown statistics	
`drawdown_episode`	count of high water mark dates since inception
`high_water_mark`	NAV at start of current drawdown episode
`drawdown`	distance of current NAV relative to most recent high water mark
`maximum_drawdown`	the maximum drawdown in the current drawdown episode
`drawdown_length`	the length, in days, of a completed drawdown episode
`drawdown_date`	the date on which an episode's maximum drawdown occured