Double the Bet Until We Win

```
doubleBet = function(n) {

  urn = c(-1, 1)

  for (i in 1:n) {
    res = sample(urn, size = 1)
    if (res > 0) return(i)
  }
  return(NA)
}
```

Add a Check

- If the caller provides input that is not numeric
- Should we:
  - Issue a warning?
  - Modify the input and continue?
  - Stop all together?

```
if (!is.numeric(numBets) {
  stop("n must be numeric")
}
```

```
doubleBet = function(n) {

  if(!is.numeric(n)) stop("n must be numeric")

  urn = c(-1, 1)

  for (i in 1:n) {
    res = sample(urn, size = 1)
    if (res > 0) return(i)
  }
  return(NA)
}
```

doubleBet(3) suppose draws -1, -1, 1

doubleBet = function(n) {														
`if(!is.numeric(n))` `stop("n must …")`	**x**													
`urn = c(-1, 1)`		**x**												
`for (i in 1:n) {`			**x**			**x**			**x**					
`res = sample(urn, 1)`				**x**			**x**			**x**				
`if (res > 0) return(i)`					**x**			**x**			**x**			
`}`														
return(NA)														
}														

doubleBet(3) suppose draws -1, -1, -1

doubleBet = function(n) {														
`if(!is.numeric(n))` `stop("n must …")`	**x**													
`urn = c(-1, 1)`		**x**												
`for (i in 1:n) {`			**x**			**x**			**x**					
`res = sample(urn, 1)`				**x**			**x**			**x**				
`if (res > 0) return(i)`					**x**			**x**			**x**			
`}`														
return(NA)												**x**		
}														

The for loop

Looping is the repeated evaluation of a statement or block of statements.

Much of what is handled using loops in other languages can be more efficiently handled in R using vectorized calculations or one of the apply mechanisms.

However, certain algorithms, such as those requiring recursion, can only be handled by loops.

There are two main looping constructs in R: `for` and `while`.

For loops

A *for loop* repeats a statement or block of statements a predefined number of times.

The syntax in R is

```
for ( var in vector ){
  statement
}
```

For each element in `vector`, the variable `var` is set to the value of that element and `statement(s)` is evaluated.

`vector` often contains integers, but can be any valid type.

The while loop

While loops

A *while loop* repeats a statement or block of statements for as many times as a particular condition is `TRUE`.

The syntax in R is

```
while (condition){
  statement
}
```

`condition` is evaluated, and if it is `TRUE`, the statement(s) is evaluated. This process continues until condition evaluates to `FALSE`.

Number of bets until win $1

Let’s use a while loop to write a function that continues to place bets (doubling each time) until we win $1.

We are interested in the number of bets it takes to win $1.

What does the `while` condition check?

A. The number of bets?

B. The winnings?

```
doubleWhile  = function(){

 bets  = 0
 urn = c(-1, 1)
 res = -1

 while (res < 0) {
  res = sample(urn,  1)
  bets = bets + 1
 }
 return(bets)
}
```

doubleWhile()
suppose draws -1, -1, 1

A. Correct
B. Wrong

doubleWhile = function() {														
bets = 0	x													
urn = c(-1, 1)		x												
res = -1			x											
while (res < 0) {				x										
res = sample(urn, 1)					x		x		x					
bets = bets + 1						x		x		x				
}														
return(bets)														
}														

doubleWhile()
suppose draws -1, 1

A. Correct
B. Wrong

doubleWhile = function() {														
bets = 0	x													
urn = c(-1, 1)		x												
res = -1			x											
while (res < 0) {				x			x			x				
res = sample(urn, 1)					x			x						
bets = bets + 1						x			x					
}														
return(bets)											x			
}														

How do we learn from our simulator?

- Run the simulation many times and examine the distribution of possible outcomes
- We might want to convert the number of bets needed to the size of the wallet needed to play this strategy

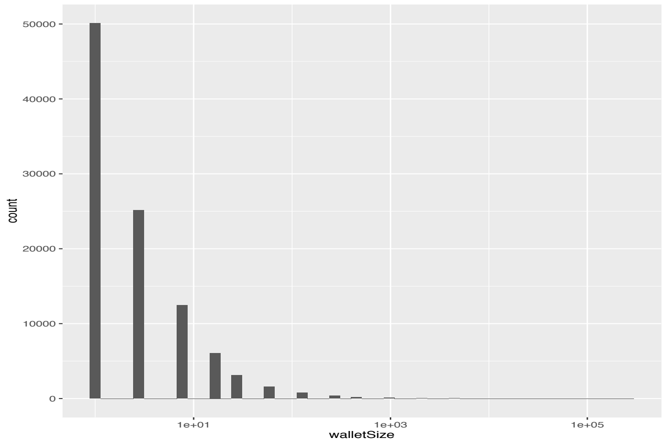
Number of Bets Until Win

```
> numBetsUntil =
     replicate(100000, double.Inf())


> summary(numBetsUntil)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.000   1.000   1.994   2.000  18.000
```

Wallet Size Required

```
walletSize = 2^numBetsUntil – 1
```



The **break** statement causes a loop to exit. This is particularly useful with while loops, which, if we're not careful, might loop indefinitely (or until we kill R).

```
doubleWhile = function(){

 res = -1
 bets = 0
 max.iter = 1000
 urn = c(-1, 1)

 while(res < 0){
  res = sample(urn, 1)
  bets = bets + 1
  if(bets > max.iter){
    warning("Maximum iteration reached")
    break
  }
 }
 return(bets)
}
```

Why don't we just call stop()?

We don't want our function to cause an error

Vector version

```
double.vec = function(n) {

  res = sample(c(-1, 1), size = n,
               replace = TRUE)

  firstWin = which(res > 0)[1]

  if (length(firstWin) == 0) return(NA)
  return(firstWin)
}
```

Which is more efficient?

`doubleBet` or
`doubleBet.vec`

```
> system.time(replicate(100000,
                      doubleBet(200)))

   user  system elapsed
  1.738   0.167   1.952

> system.time(replicate(100000,
                   doubleBet.vec(200)))

   user  system elapsed
  1.906   0.138   2.063
```

Why are the timings so similar?

The for loop version often stops after a few samples, but the vector version always takes all n samples