Section 7: Efficient Simulation in R.

Ensure you have completed all previous worksheets before advancing to this one.

1 Measuring Computation Time and Tips

For simple speed comparisons we can use `system.time()`, for multiple lines of code wrap using `{ }`. Try the following:

```
> system.time(dnorm(2))
> system.time(dnorm(2); dnorm(3))
> system.time({dnorm(2); dnorm(3)})
```

Note if you had put different commands on different lines in an R script, you would not have needed the ; symbol in the last example.

The output gives user/system/elapsed times. If system is a large percentage of elapsed time this indicates heavy Operating System (OS) management and possibly wasteful memory accesses.

**Tip 1**: Writes to data frames are **slow**!

```
> x=matrix(rep(0,10000),ncol=1)
> y=data.frame(V1=rep(0,10000))
```

**Task**: How long does the following code take to run?

```
> for(i in 1:10000) { x[i,1] = 1 }
> for(i in 1:10000) { y$V1[i] = 1 }
> ```
Tip 2: When reading data frames, don’t use the notation convention for a matrix type object. How long does the following code take?

```r
> for(i in 1:10000) { a = y[i,1] }
```


```r
> for(i in 1:10000) { a = y$V1[i] }
```

Tip 3: Pre-allocate all the memory you will need: don’t expand vectors/matrices/data frames on the fly. Compare the speed differences below:

```r
> x=NULL
> for(i in 1:10000) { x = rbind(x, rnorm(20)) }
```

```r
> x=NULL
> for(i in 1:10000) { x[i] = rnorm(1) } }
```

```r
> x=matrix(nrow=10000,ncol=20)
> for(i in 1:10000) { x[i,] = rnorm(20) }
```

```r
> x = vector("numeric", 10000)
> for(i in 1:10000) { x[i] = rnorm(1) }
```

Tip 4: Ruthlessly vectorize everything possible - for loops are evil!

```r
> x = seq(-6, 6, length.out=1e5)
> area = 0
```

**Task:** How long does the following code take to run?

Method 1:

```r
> for(i in 2:1e5) { area = area+dnorm(x[i])*(x[i]-x[i-1]) }
```
Method 2:

```r
> sum((x[2:100000]-x[1:99999])*dnorm(x[2:100000]))
```

> 

2 2D Random Walk Example

Consider the following methods to perform the same simulation of a random walk in 2 dimensions (use an R script).

**Version A:** Naive Coding

```r
rand_walk_A = function(n) {
    walk = data.frame(x = c(0), y = c(0))
    for(i in 2:n) {
        if(sample(c(TRUE, FALSE), 1)) {
            walk[i,1] = walk[i-1,1] + sample(c(-1,1), 1)
            walk[i,2] = walk[i-1,2]
        } else {
            walk[i,1] = walk[i-1,1]
            walk[i,2] = walk[i-1,2] + sample(c(-1,1), 1)
        }
    }
    walk
}
```

**Version B:** Eliminate use of data frames

```r
rand_walk_B = function(n) {
    x = 0; y = 0
    for(i in 2:n) {
        if(sample(c(TRUE, FALSE), 1)) {
            x[i] = x[i-1] + sample(c(-1,1), 1)
            y[i] = y[i-1]
        } else {
            x[i] = x[i-1]
            y[i] = y[i-1] + sample(c(-1,1), 1)
        }
    }
    list(x = x, y = y)
}
```


} else { x[i] = x[i-1]  
y[i] = y[i-1] + sample(c(-1,1), 1)
} list(x=x, y=y}

**Version C:** Now eliminate use of dynamically expanded memory

```r
rand_walk_C = function(n) {
  x = vector("numeric", n); y = vector("numeric", n)
  for(i in 2:n) {
    if(sample(c(TRUE, FALSE), 1)) {
      x[i] = x[i-1] + sample(c(-1,1), 1)
      y[i] = y[i-1]
    } else { x[i] = x[i-1]
      y[i] = y[i-1] + sample(c(-1,1), 1)
    } list(x=x, y=y)
}
```

**Version D:** Now vectorize

```r
rand_walk_D = function(n) {
  move = sample(1:4, n-1, replace=TRUE)
  x = c(0, cumsum(c(-1,1,0,0)[move]))
  y = c(0, cumsum(c(0,0,-1,1)[move]))
  list(x=x, y=y)
}
```

**Task:** All four functions `rand_walk_A(n)` to `rand_walk_D(n)`, where `n` is an integer to be supplied by the user, perform the same task of simulating a 2 dimensional random walk. How long does each take to perform a simulation for `n=10000`? For `n=20000`? Was this a doubling of the time taken?

>  

>  

>