Solutions for ST340 Lab 1

2019–20

1: Implement bubblesort

`a` is a vector and the function should return `a` in increasing sorted order. Example: if `a = c(3,5,2,4,1)`, then the output should be `c(1,2,3,4,5)`.

(a) Complete the function.

```
bubble.sort <- function(a) {
  n <- length(a)
  if (n == 1) return(a)
  okay <- FALSE
  while (!okay) {

    okay <- TRUE

    for (i in 1:(n-1)) {
      if (a[i] > a[i+1]) {
        tmp <- a[i]
        a[i] <- a[i+1]
        a[i+1] <- tmp
        okay <- FALSE
      }
    }
  }
  return(a)
}
```

(b) Test that it works.

```
print(bubble.sort(c(3,5,2,4,1)))
```

```
## [1] 1 2 3 4 5
```

```
print(bubble.sort(c(4,2,7,6,4)))
```

```
## [1] 2 4 4 6 7
```

(c) Look at `?system.time`.

(d) How long does it take to sort `(1,2,...,10000)`?

```
system.time(bubble.sort(1:10000))
```

```
##    user  system elapsed
##   0.001   0.000   0.002
```

(e) How about `(10000,1,2,3,...,9999)`?

```
system.time(bubble.sort(c(10000,1:9999)))
```

```
##    user  system elapsed
##   0.004   0.000   0.004
```

(f) How about `(2,3,...,2000,1)`?

```
system.time(bubble.sort(c(2:2000,1)))
```

```
##    user  system elapsed 
##   0.200   0.000   0.203
```

(g) How about a random permutation (see `?sample`) of 1,. . . ,2000?

```
system.time(bubble.sort(sample(2000)))
```

```
##    user  system elapsed 
##   0.339   0.008   0.347
```

(h) Finally, recall the worst case input is `(n,n-1,...,2,1)`. Try the worst case input with `n = 2000`.

```
system.time(bubble.sort(2000:1))
```

```
##    user  system elapsed 
##   0.474   0.000   0.476
```

2: Implement quicksort

First, increase the maximum number of nested expressions that can be evaluated.

```
options(expressions=100000)
```

`a` is a vector and the function should return `a` in increasing sorted order. Example: if `a = c(3,5,2,4,1)`, then the output should be `c(1,2,3,4,5)`.

(a) Complete the function.

```
qsort <- function(a) {
  if (length(a) > 1) {
    pivot <- a[1]
    l <- a[a<pivot]
    e <- a[a==pivot]
    g <- a[a>pivot]
    a <- c(qsort(l),e,qsort(g))  }
  return(a)
}
```

(b) Test that it works.

```
    print(qsort(c(3,5,2,4,1)))
```

```
## [1] 1 2 3 4 5
```

```
    print(qsort(c(4,2,7,6,4)))
```

```
## [1] 2 4 4 6 7
```

(c) How long does it take to quicksort `(1,2,...,2000)`?

```
system.time(qsort(1:2000))
```

```
##    user  system elapsed 
##   0.037   0.013   0.051
```

(d) How long does it take to quicksort `(2000,1999,...,1)`?

```
system.time(qsort(2000:1))
```

```
##    user  system elapsed 
##   0.035   0.007   0.043
```

(e) How long does it take to quicksort a random permutation of `(1,2,...,2000)`?

```
system.time(qsort(sample(2000)))
```

```
##    user  system elapsed 
##   0.004   0.000   0.004
```

3: Implement randomized quicksort

`a` is a vector and the function should return `a` in increasing sorted order. Example: if `a = c(3,5,2,4,1)`, then the output should be `c(1,2,3,4,5)`.

(a) Complete the function.

```
randomized.qsort <- function(a) {
  n <- length(a)
  if (n > 1) {
    pivot <- a[sample(n,size=1)]
    l <- a[a<pivot]
    e <- a[a==pivot]
    g <- a[a>pivot]
    a <- c(randomized.qsort(l),e,randomized.qsort(g))  }
  return(a)
}
```

(b) Test that it works.

```
    print(randomized.qsort(c(3,5,2,4,1)))
```

```
## [1] 1 2 3 4 5
```

```
    print(randomized.qsort(c(4,2,7,6,4)))
```

```
## [1] 2 4 4 6 7
```

(c) How long does it take to sort `(1,2,...,2000)`, `(2000,1999,...,1)`, or a random permutation, using randomized quicksort?

```
system.time(randomized.qsort(1:2000))
```

```
##    user  system elapsed 
##    0.01    0.00    0.01
```

```
system.time(randomized.qsort(2000:1))
```

```
##    user  system elapsed 
##   0.009   0.001   0.009
```

```
system.time(randomized.qsort(sample(2000)))
```

```
##    user  system elapsed 
##   0.009   0.000   0.009
```

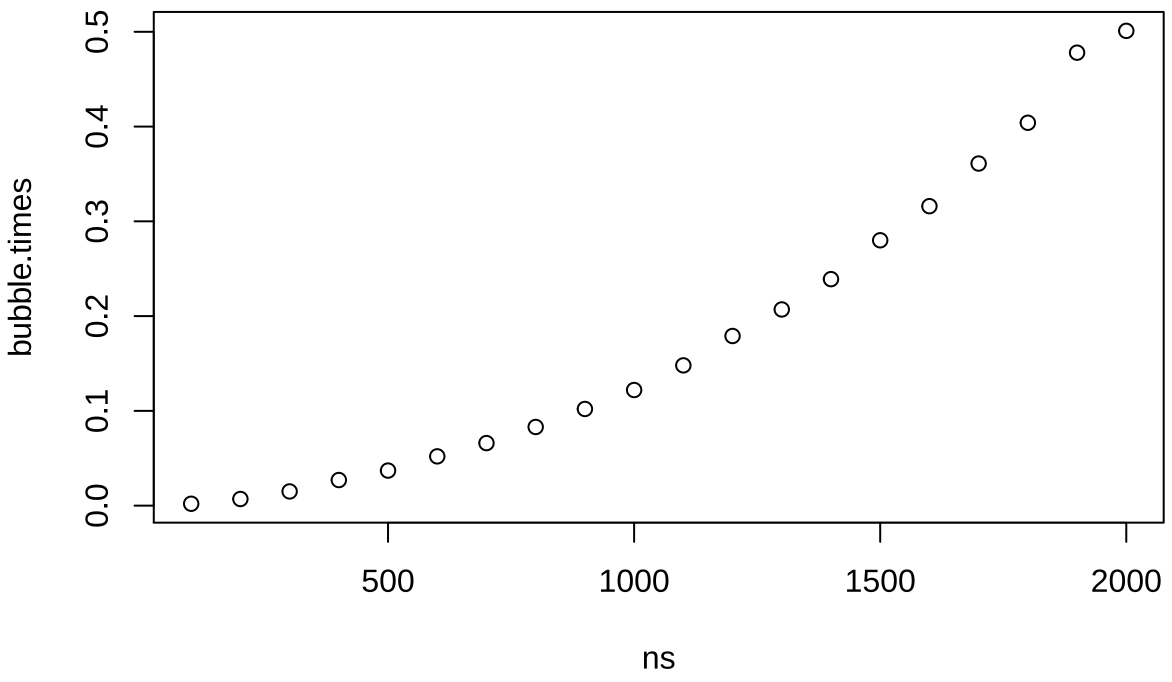
4: Compare the running time of the algorithms

Worst-case bubble and quicksort:

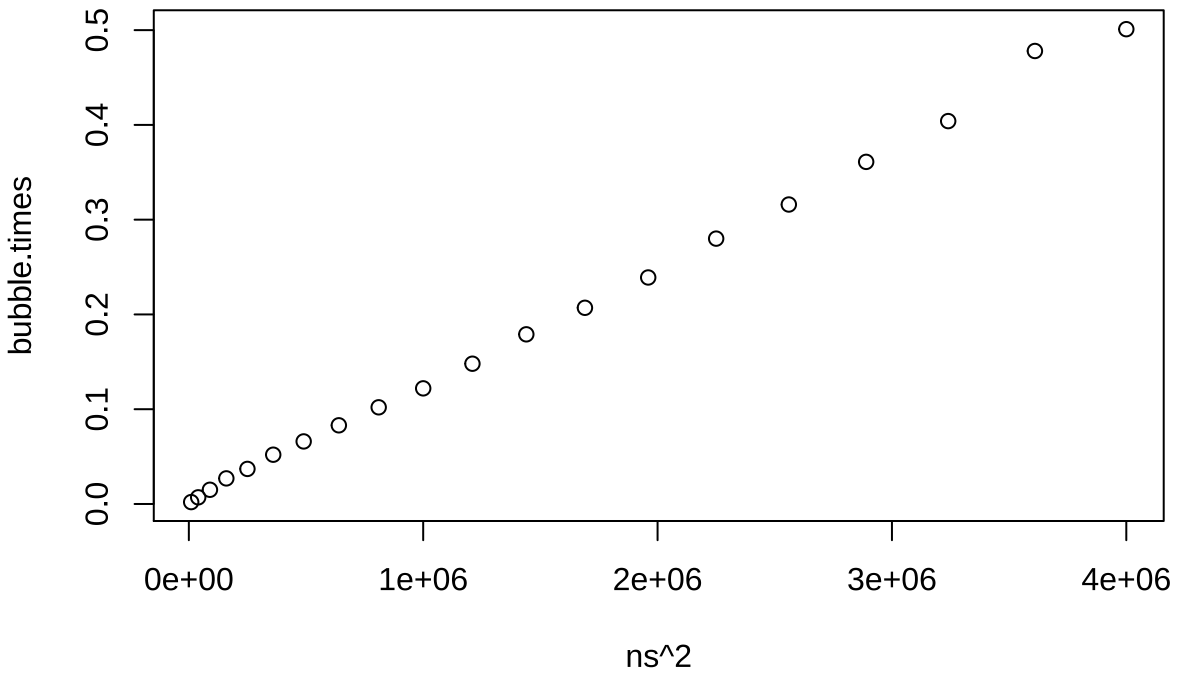
```r
ns <- seq(from=100,to=2000,by=100)
bubble.times <- rep(0,length(ns))
quick.times <- rep(0,length(ns))
randomized.quick.times <- rep(0,length(ns))
for (i in 1:length(ns)) {
  a <- ns[i]:1 # a is in reverse sorted order
  bubble.times[i] <- system.time(bubble.sort(a))[3]
  quick.times[i] <- system.time(qsort(a))[3]
  randomized.quick.times[i] <- system.time(randomized.qsort(a))[3]
}
```

(a) Plot `bubble.times` against `ns`, and also against `ns^2`.

```r
plot(ns,bubble.times)
```



```r
plot(ns^2,bubble.times)
```



Note: The `system.time()` command has poor rounding properties on Windows machines; your plots might look less clear than these. An alternative in Windows is to use `Sys.time()` instead, which just records the current date and time. You can take the difference of the return value of this function at the beginning and the end of the period to be timed. For example:

```
t.start <- Sys.time()
system.time(qsort(1:2000))
```

```
##    user  system elapsed
##   0.038   0.004   0.045
```

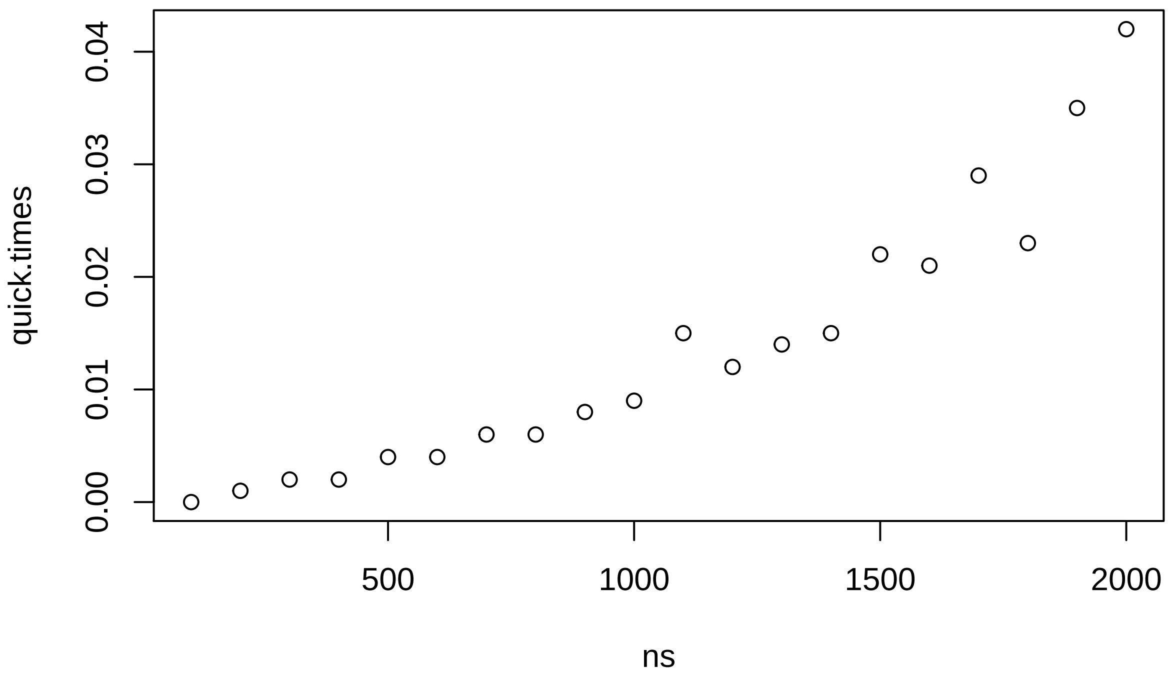
```
t.end <- Sys.time()
print(t.end-t.start)
```

```
## Time difference of 0.08222008 secs
```

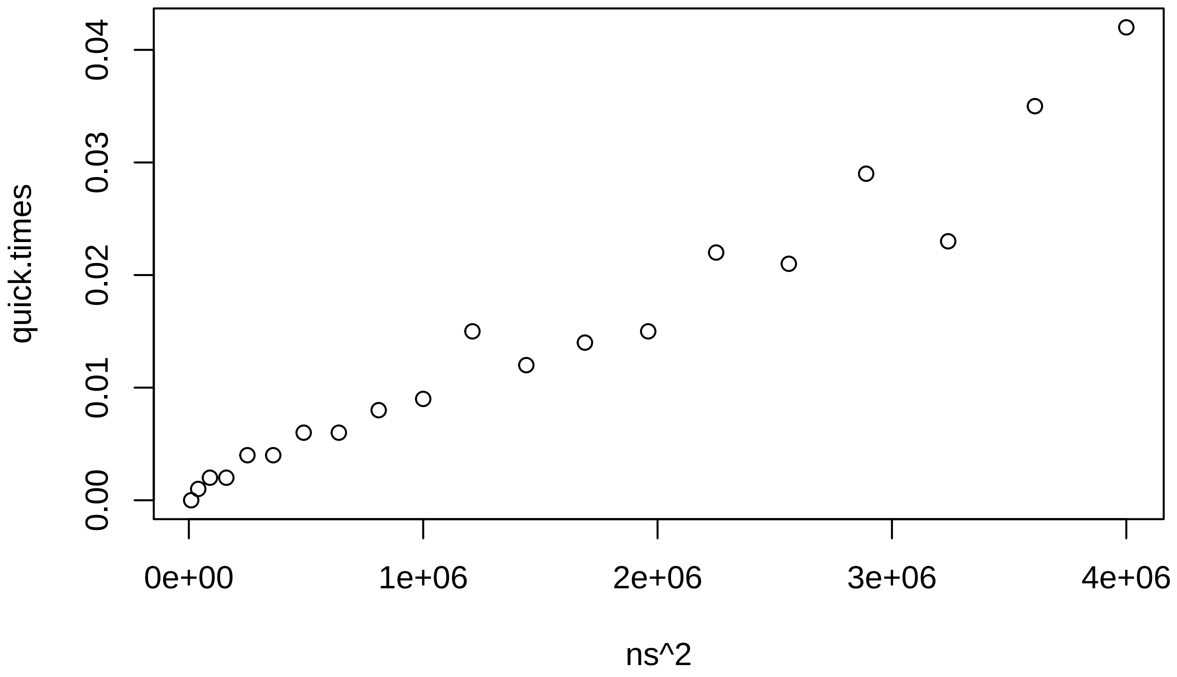
The correspondence between the two timing methods is not fantastic.

(b) Plot `quick.times` against `ns`, and also against `ns^2`.

```
plot(ns,quick.times)
```



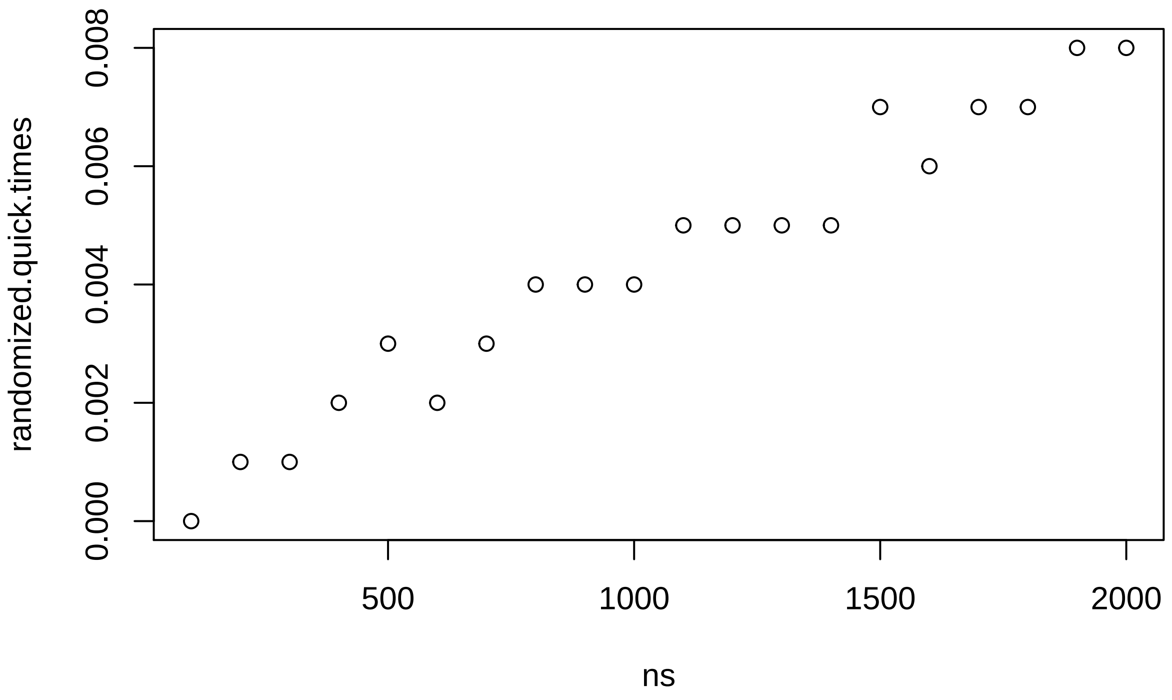
```
plot(ns^2,quick.times)
```



The approximately linear growth of worst case running time with squared input size is consistent with the theoretical $O(n^2)$ time complexity of both of these algorithms (contrast the algorithms' y-axes, however).

(c) Plot `randomized.quick.times` against `ns`.

```
plot(ns,randomized.quick.times)
```



(Notice that the limited precision of `system.time` is probably evident here; the actual running time is obviously not jumping up in steps.)

5: Implement counting sort

`a` is a vector of positive integers and the function should return `a` in increasing sorted order. Example: if `a = c(3,5,2,4,1)`, then the output should be `c(1,2,3,4,5)`.

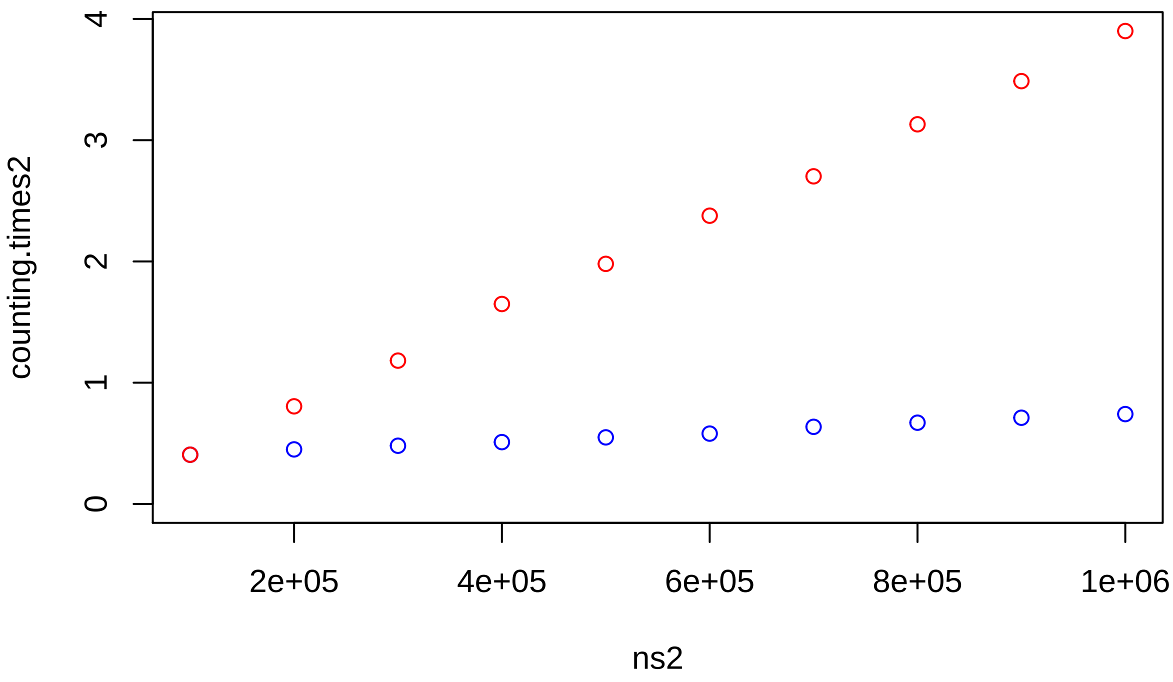
```
countingsort <- function(a) {
  n <- length(a); N <- max(a)
  c <- rep(0,N)
  for (i in 1:n) {
    c[a[i]] <- c[a[i]] + 1
  }
  b <- rep(0,n)
  i <- 1
  for (j in 1:N) {
    if (c[j] > 0) {
      for (k in 1:c[j]) {
        b[i] <- j; i <- i+1
      }
    }
  }

  return(b)
}
```

6: Compare the running time of randomized quick sort and counting sort

```r
N <- 1e7 # maximum value of the positive integers
ns2 <- 1e5*(1:10)
randomized.quick.times2 <- rep(0,length(ns2))
counting.times2 <- rep(0,length(ns2))
for (i in 1:length(ns2)) {
  # each element of a is a draw from a categorical distribution
  a <- sample(N,size=ns2[i],replace=TRUE)
  counting.times2[i] <- system.time(countingsort(a))[3]
  randomized.quick.times2[i] <- system.time(randomized.qsort(a))[3]
}
```

(a) Plot `counting.times2` against `ns2`.
(b) Add `randomized.quick.times2` against `ns2` to the same plot.

```r
yu <- max(max(counting.times2),max(randomized.quick.times2))
plot(ns2,counting.times2,ylim=c(0,yu),col="blue")
points(ns2,randomized.quick.times2,col="red")
```



This plot is consistent with the fact that both algorithms are $O(n)$ for this type of input.

(c) How would you describe the time complexity of randomized quick sort for the type of inputs generated above, assuming we only change `n`?

Answer: randomized quick sort is now in $O(n \times N) = O(n)$ in the worst case since we can only have N distinct pivots with N a constant. If N was very very large, we would observe "$O(n \log n)$ type" behaviour for smaller values of n but the algorithm would still be $O(n)$.

(d) Does this contradict the $\Omega(n \log n)$ lower bound discussed in class for comparison-based sorting algorithms?

Answer: While the algorithm will run in $O(n)$ time on this type of input, it does not contradict the result since the $\Omega(n \log n)$ bound applies to the worst case for any possible set of inputs, while we are only considering a restricted set of "easy" inputs here.