ST340 Lab 1: Time complexity

2020–21

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)`.

The pseudo code for the bubble sort is given in slide 15 of lecture slides 1. The worst case running time for the bubble sort is $\Theta(n^2)$, which means the running time is proportional to n^2

```
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]){
        x = a[i+1]
        a[i+1] = a[i]
        a[i]   = x
        okay = FALSE
        }

      }
  }
  return(a)
}
```

(a) Complete the function above.

(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)`?

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

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

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

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

```
##    user  system elapsed
##       0       0       0
```

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

```
##    user  system elapsed
##    0.02    0.00    0.02
```

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

```
##    user  system elapsed
##    0.25    0.00    0.25
```

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

```
##    user  system elapsed
##     0.4     0.0     0.4
```

(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
##    0.69    0.00    0.69
```

2: Implement quicksort

The worst case number of comparisons is in $\Theta(n^2)$

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)`.

```
qsort <- function(a) {
  if (length(a) > 1) {
    pivot <- a[1]
    less    = a[a < pivot]
    equals  = a[a == pivot]
    greater = a[a > pivot]
    a = c(qsort(less),equals,qsort(greater))
  }
  return(a)
}
```

(a) Complete the function above.

(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
```

For this algorithm I am pretty sure the time complexity is the same whether the sequence is sorted in increasing or decreasing order. But the worst case is when the sequence is sorted. So we can see in (c) and (d) that the `system.time()` is very similar.

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

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

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

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

```
##    user  system elapsed
##    0.11    0.03    0.14
```

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

```
##    user  system elapsed
##    0.05    0.02    0.06
```

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

```
##    user  system elapsed
##       0       0       0
```

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)`.

```
randomized.qsort <- function(a) {
  n <- length(a)
  if (n > 1) {
    pivot <- a[sample(n,size=1)]
    less    = a[a < pivot]
    equals  = a[a == pivot]
    greater = a[a > pivot]
    a = c(randomized.qsort(less),equals,randomized.qsort(greater))

  }
  return(a)
}
```

(a) Complete the function above.

(b) Test that it works.

```
print(randomized.qsort(c(3,5,2,4,1)))
print(randomized.qsort(c(4,2,7,6,4)))
```

The worst case **expected** number of comparisons is in $\mathcal{O}(nlogn)$

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

4: Compare the running time of the algorithms

Worst-case bubble and quicksort:

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

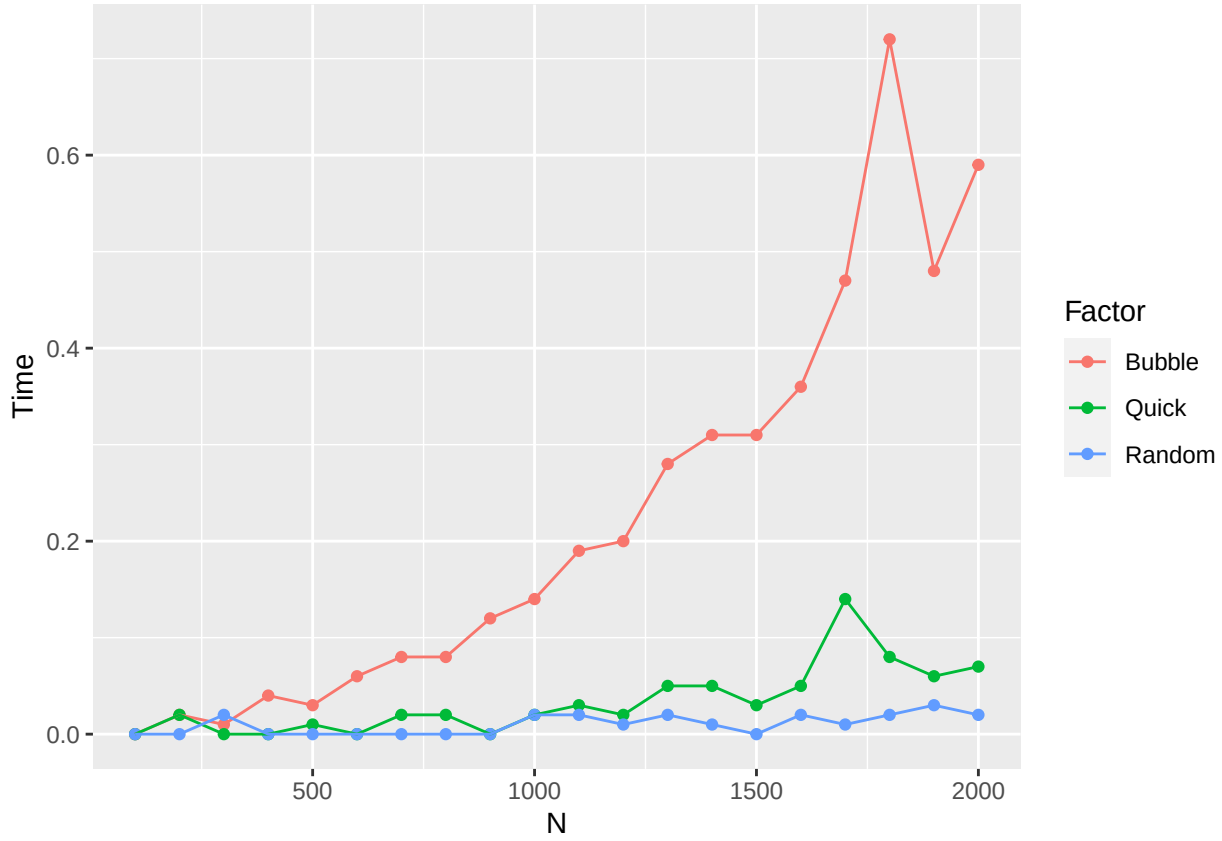
I have plotted the algorithms `system.time()` against the length of the input sequence `N`

```
algo_times = tibble('N' = ns,
                    'N squared' = ns^2,
                    'Bubble'    = bubble.times,
                    'Quick'     = quick.times,
                    'Random'    = randomized.quick.times)

algo_times = algo_times %>%
                        pivot_longer(c(Bubble,Quick,Random), names_to = "Factor", values_to = "Time")



ggplot(algo_times, aes(N, Time, color = Factor)) +
  geom_point() +
  geom_line()
```

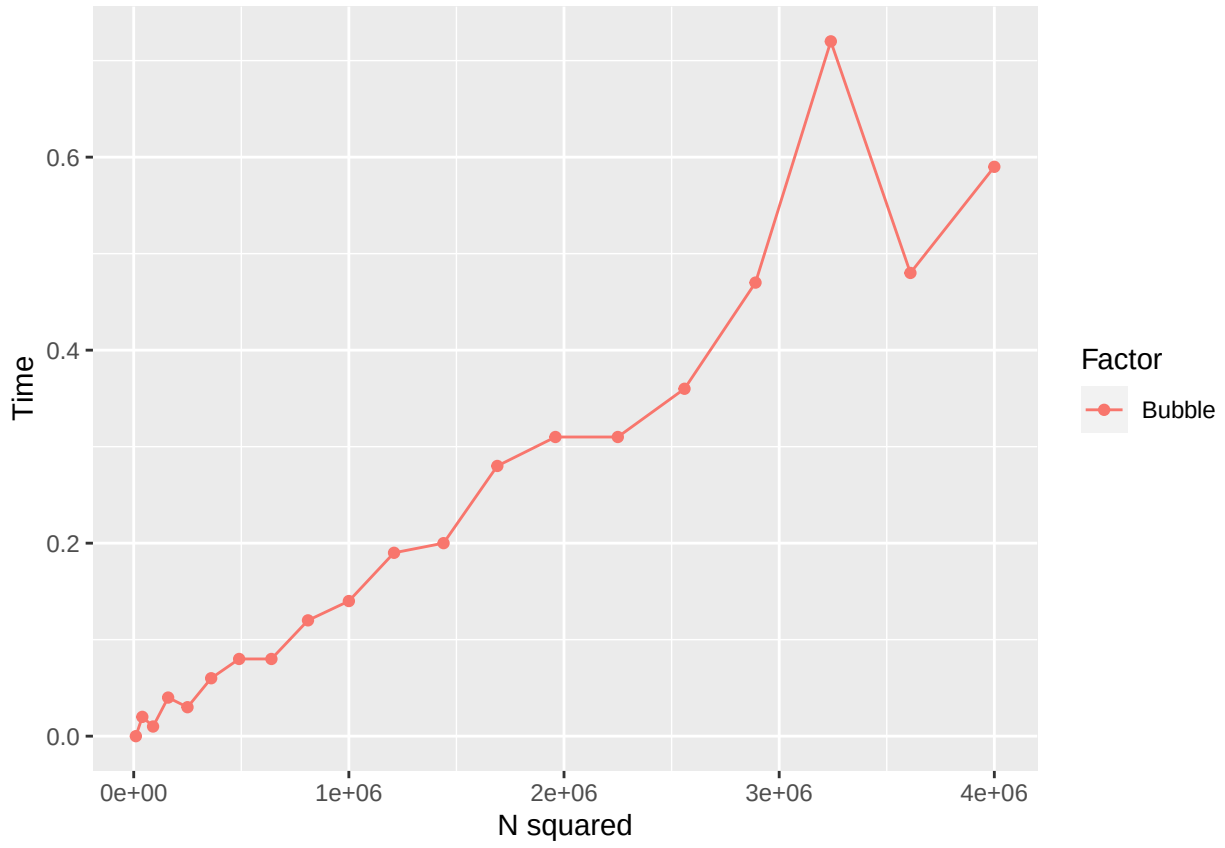


I am not sure what he wants to plot for the N squares as the system_time will be very long. I also get the error: `Error: C stack usage  15924640 is too close to the limit Timing stopped at: 0.95 0 0.96`

I understand now, so we want to plot the corresponding system times of the Quick sort and bubble sort algorithms generated from vectors of length `ns` against `ns squared`. This is because the worst case running time of bothh these algorithms is in $\Theta(n^2)$. Therefore we should see a linear relationship in the plots.

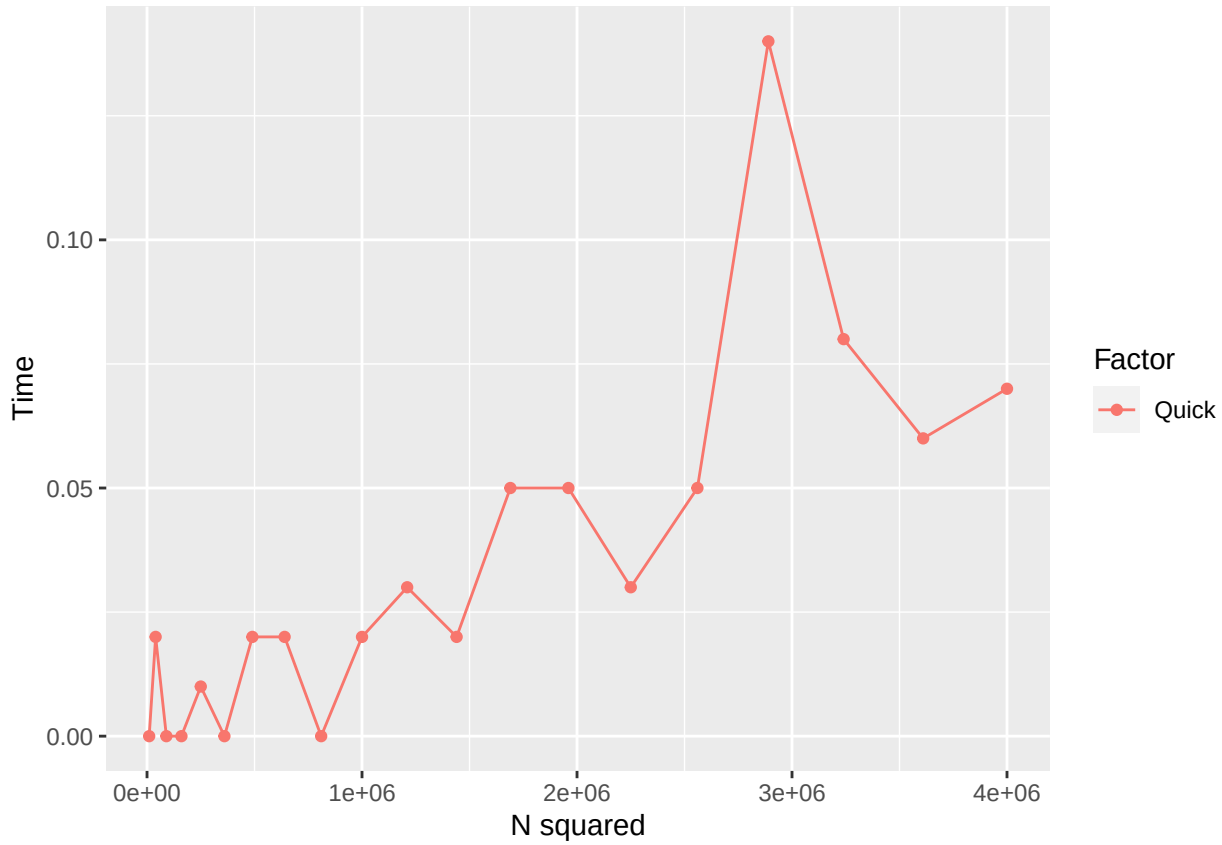
```
algo_times %>%
  filter(Factor == 'Bubble') %>%

ggplot(aes(`N squared`, Time, color = Factor)) +
  geom_point() +
  geom_line()
```




```
algo_times %>%
  filter(Factor == 'Quick') %>%

ggplot(aes(`N squared`, Time, color = Factor)) +
  geom_point() +
  geom_line()
```


Time
0.10
0.05
0.00
0e+00
1e+06
2e+06
3e+06
4e+06
N squared
Factor
Quick

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=c*(1:N)
  b[b>0]

  return(b)
}
```

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

```
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]
}
```

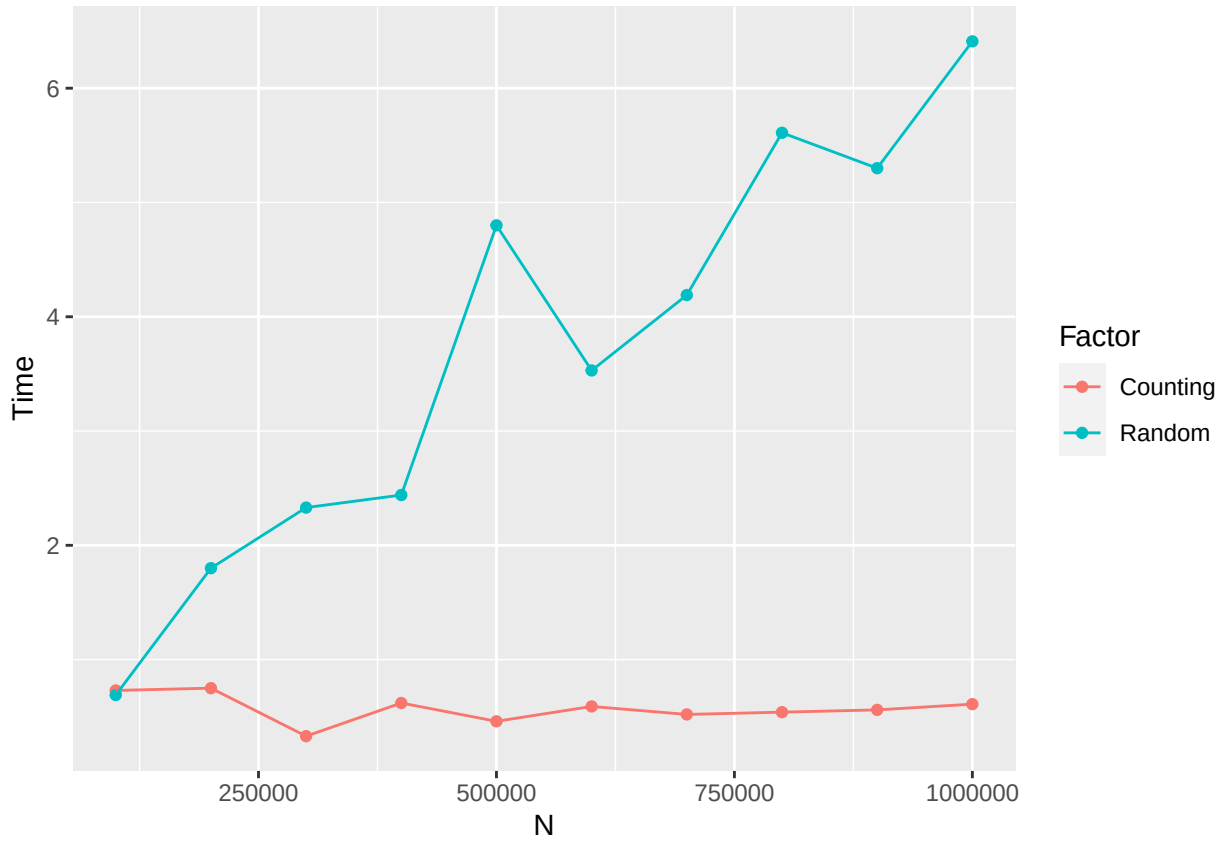
(a) Plot `counting.times2` against `ns2`.
(b) Add `randomized.quick.times2` against `ns2` to the same plot.

```
algo_times_2 = tibble('N' = ns2,
                      'Counting'  =  counting.times2,
                      'Random'    = randomized.quick.times2)

algo_times_2 = algo_times_2 %>%
                       pivot_longer(c(Counting, Random), names_to = "Factor", values_to = "Time")


fun.1 = function(x) x*log(x)

 ggplot(algo_times_2, aes(N, Time, color = Factor)) +
      geom_point() +
      geom_line()
```



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

There seems to be a linear relationship between N and the time complexity of the randomized quick sort algorithm.

So we are keeping N constant therefore the randomized quick sort is now in $\mathcal{O}(n \times N) = \mathcal{O}(n)$ in the worst case since the maximum number in the vector is now bounded by N.

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

So in lectures we are given the **Theorem**: Any comparison-based sorting algorithm A makes $\Omega(nlog(n))$ comparisons in the worst case.

So there exist $M > 0$ and $n_0 > 0$ $s.t$ $f(n) \geq M[nlog(n)]$ where $f(n)$ is the worst case number of comparisons.

So the algorithm will run in $\mathcal{O}(n)$ time on this type of input but we are only considering a restricted type of input, and the lower bound from the theorem applies to the worst case for any possible set of inputs.