ST340 Lab 1: Time complexity

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)`.

```
bubble.sort <- function(a) {
  n <- length(a)
  if (n == 1) return(a)
  okay <- FALSE
  while (!okay) {
    # implement the pseudo-code in the lecture slides
    okay <- TRUE
  }
  return(a)
}
```

(a) Complete the function above.

(b) Test that it works.

```
print(bubble.sort(c(3,5,2,4,1)))
print(bubble.sort(c(4,2,7,6,4)))
```

(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?

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

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)`.

```
qsort <- function(a) {
  if (length(a) > 1) {
    pivot <- a[1]
    # implement the pseudo-code in the lecture slides
  }
  return(a)
}
```

(a) Complete the function above.

(b) Test that it works.

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

(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)`?

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)]
    # implement the pseudo-code in the lecture slides
  }
  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)))
```

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

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`.

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)
  # implement the pseudo-code in the lecture slides
  return(NULL)
}
```

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.
(c) How would you describe the time complexity of randomized quick sort for the type of inputs generated above, assuming we only change `n`?
(d) Does this contradict the $\Omega(n \log n)$ lower bound discussed in class for comparison-based sorting algorithms?