Introductory Computer Programming Assignment 2

Soumya Mukherjee

12 January 2020

1. Random Number Generator to generate random numbers 0-99

We are given a random number generator `rcoin()` that generates either 0 or 1 with equal probability $\frac{1}{2}$, every time it is called. The outcomes of different calls are independent of each other. We want to write an algorithm `r100()` using `rcoin()` that produces a random integer between 0 and 99 (both inclusive).

Procedure :

- Each time `r100()` is called, the random number generating algorithm `rcoin()` will be called N times (where the choice of N is explained later).
- The output will be a sequence of 0's and 1's stored in an array of length N, with the first number generated being the last number of the sequence and so on. This sequence will be treated as a N-digit binary number. Suppose under `r100()` , the algorithm `rcoin()` will be invoked 3 times and the outcomes are $0, 1, 1$ respectively. Then the 3-digit binary number is $(110)_2$
- Using this procedure a random integer X between 0 and $2^N - 1$ (both inclusive) is chosen since maximum N-digit binary is $(\underbrace{111...1}_{N\ times})_2 = (2^N - 1)_{10}$
- We convert the output X from binary to decimal.
- Let $(2^N - 1) = 100 \times q + r, \quad 0 \leq r \leq 99, \;\; q, r \in \mathbb{Z}$. If $X \geq 100 \times q$, then reject the number and run `r100()` once again. If $X < 100 \times q$, then consider the remainder O of X after dividing by 100 i.e. last two digits of the decimal representation of X.
- In each call to `r100()`, the probability of rejection of X (the decimal version) under this procedure is $\frac{r+1}{2^N}$. Since each call to `r100()` is independent of each other, the probability that k calls to `r100()` is unable to produce a random integer between 0 and 99 (both inclusive) is $(\frac{r+1}{2^N})^k$ which tends to zero as k tends to ∞ increases with N fixed. Thus the probability that `r100()` produces a desired random integer is

  $$1 - \lim_{k \to \infty} (\frac{r+1}{2^N})^k = 1 - 0 = 1 \tag{1}$$

  However the algorithm is not guaranteed to terminate in a finite number of steps. This is a randomized algorithm, more specifically a Las Vegas algorithm.
- The number of calls to rcoin is a random variable $C = N \times Y$ where Y follows a geometric distribution defined on $\{1, 2, ...\}$ with pmf $f(y) = p \times (1-p)^{y-1} \times I_{\{1,2,..\}}$ where $I_{\{1,2,..\}}$ is the indicator function of the set $\{1, 2, ...\}$ and $p = 1 - \frac{r+1}{2^N}$. The expected number of calls to `r(100)` is $E(Y) = \frac{1}{p} = \frac{1}{1-\frac{r+1}{2^N}}$. Hence the expected number of calls to `rcoin()` is $E(C) = \frac{N}{1-\frac{r+1}{2^N}}$.Since the number of calls to `rcoin()` determines the number of steps required to get the desired output, N is so chosen that the expected number of calls to `rcoin` is minimized. Consider the following table

Table 1: Expected Number of calls to rcoin()

N	Largest Binary number generated and converted Decimal number	No.s Accepted	No.s Rejected	Expected No. of calls to rcoin()
7	$(1111111)_2 = (127)_{10}$	0-99	100-127	8.96
8	$(11111111)_2 = (255)_{10}$	0-199	200-255	10.24
9	$(111111111)_2 = (511)_{10}$	0-499	500-511	9.216
10	$(1111111111)_2 = (1023)_{10}$	0-999	1000-1023	10.24
11	$(11111111111)_2 = (2047)_{10}$	0-1999	2000-2047	11.264
12	$(111111111111)_2 = (4095)_{10}$	0-3999	4000-4095	12.288
13	$(1111111111111)_2 = (8191)_{10}$	0-7099	8100-8191	13.147
14	$(11111111111111)_2 = (16383)_{10}$	0-16299	16300-16383	14.072
15	$(111111111111111)_2 = (32767)_{10}$	0-32699	32700-32767	15.032

As N becomes arbitrarily large, $E(C) \approx N$ as the denominator of $E(C)$ becomes almost equal to 1. We observe this also from the previous table. Hence it is not necessary to calculate $E(C)$ for values higher than $N = 15$. From the table we see that $E(C)$ is minimum for $N = 7$. Hence we choose N to be 7. Hence we have $p = 1 - \frac{27+1}{2^7} = 0.78125$ and

$$E(C) = \frac{7}{0.78125} = 8.96 \tag{2}$$

The probability that an integer in the desired range is chosen is $1 - \frac{r+1}{2^N}$. Again $P(X = i) = \frac{q}{2^N}$ as, X assumes value i only when the numbers $i, 100+i, 1000+i, ..., 100 \times (q-1)+i$ are chosen, $\quad \forall \quad i = 0, 1, 2, ..., 2^N - 1$.

If O be the generated random number then

$$P(O = i) = \begin{cases} \frac{\frac{1}{2^N}}{1-\frac{r+1}{2^N}} = \frac{q}{100 \times q} = \frac{1}{100} \ \forall \ i = 0, 1, 2, ..., 99 \\ 0 \ , \text{otherwise} \end{cases} \tag{3}$$

Hence the generated random number

$$O \sim Uniform\{0, 1, 2, ..., 99\}$$

Algorithm :

```
r100()

x = 0
q = floor((2^N - 1)/100)
for (i = 1 to N)
{
 x = x + rcoin()*(2^(i-1))
}
if ( x >= 100*q )
  r100()
else
{
  o = x - 100*(floor(x/100))
  print o
}
```

Correctness :

For proving the correctness of the algorithm, we need to prove that-

1. When the algorithm terminates, the algorithm generates an integer between 0 and 99 (both inclusive).
2. When the algorithm terminates, it generates a random integer between 0 and 99 (both inclusive) i.e. the probability distribution of the generated number O is $Uniform\{0, 1, 2, ..., 99\}$.
3. The algorithm terminates with probability 1.

For proving the first part, we need to prove the following loop invariant for the for loop in the algorithm.

Statement:

> At the beginning of ith iteration of for loop, the random number generated (in decimal) is less than or equal to $2^{i-1} - 1$

Initialisation:

- `rcoin()` is not called before starting the for loop for $i = 1$.
- x represents the decimal version of the random number generated by `rcoin()`.
- So $x = 0$ and $x \leq 2^{i-1} - 1 = 0$
- Therefore the loop invariant is true for $i = 1$.

Maintenance:

- At the beginning of the ith iteration of for loop , `rcoin()` has been invoked $i-1$ times and $x \leq 2^{i-1}-1$.
- In the ith iteration of the for loop, `rcoin()` is invoked only once.
- The outcome of `rcoin()` in the ith iteration of the for loop, say X_i will be either 0 or 1, which is mutiplied by 2^{i-1} added to x and again stored into x itself. Therefore,

$$\underbrace{x + X_i \times 2^{i-1}}_{x} \leq \underbrace{2^{i-1} - 1 + 2^{i-1}}_{2^i - 1}$$

- So, the above loop invariant is true for index $i + 1$.

Termination:

- The for loop essentially increments i by 1 every time before it runs.
- The loop terminates when $i > N$.
- As each loop iteration increases i by 1, we must have $i = N + 1$ at that time.
- Substituting $N + 1$ for i in the loop invariant, we have that the generated random number $x \leq 2^N - 1$.
- Hence, the loop invariance is true.

For proving the second part, we need to prove that when the algorithm terminates, it generates a random integer between 0 and 99 (both inclusive) i.e. the probability distribution of the generated number O is $Uniform\{0, 1, 2, ..., 99\}$ as shown in (3)

For proving the third part, we need to prove that the algorithm terminates with probability 1, which we have already shown in (1).

Hence we have proved that the algorithm is correct.

Expected Number of times `rcoin()` is invoked:

The algorithm proposed is a Las Vegas randomized algorithm which always gives the correct output but its runtime is not fixed. The runtime is a random variable since the number of times `rcoin()` is called is a random variable. The expected number of times `rcoin()` is invoked is $E(C) = 8.96$ as shown in (2).

2. Quicksort Algorithm using Lomuto Paritioning Scheme:

Consider an array $A[1, 2, ..., n]$ of length n which will be sorted using Quicksort alogrithm. In this algorithm the non-randomised Lomuto scheme will be used as the patitioning scheme.

The sorting algorithm is as follows:

```
QUICKSORT(A,p,r)

if (p<r) {
  q = PARTITION(A,p,r)
  QUICKSORT(A,p,q-1)
  QUICKSORT(A,q+1,r)
}

PARTITION(A,p,r)

x = A[r]
i = p-1
for (j = p,p+1,...,r) {
  if ( A[j] <= x) {
    i = i+1
    swap( A[i], A[j])
  }
}
swap(A[i+1], A[r])
return i+1
```

The functions `QUICKSORT` and `PARTITION` have three arguments. The first argument `A` stands for the array , `p` and `q` represent the starting and ending position of the array or sub-array (depending on different situations).

Let $T(n)$ be number of comparisons made by the non-randomized quicksort algorithm using the Lomuto partitioning scheme with input array of length n. It is quite clear that the runtime of `PARTITION(A,p,r)` is linear in the length of input array. If the array is already sorted then the comparison "$A[j] <= x$" and the swapping "$swap(A[i], A[j])$" will be done exactly one less number of times that of length of array or sub-array. Hence running time of `PARTITION` is $\Theta(n)$ with input array (sub-array) of length n.

As, the array is already sorted `PARTITION(A,p,q)` will always return the last position q as the final position of the pivot, resulting in the right sub array $A[q+1, ..., r]$ being empty and left sub-array $A[p, ..., q-1]$ being left with exactly one element less than that of input sub-array . Hence the recursion relation of runtime of `QUICKSORT` with input array $A[1, 2, ..., n]$ is

$$T(n) = T(n-1) + \Theta(n)$$

$$\implies T(n) = \Theta(n^2)$$

3. Finding Order Statistic using Quicksort:

The quicksort algorithm can be modified as follows to compute a specific order statistic (i.e. quantile) in linear time :

After the partition step, the recursion is called only on one of the subarrays, namely the one containing the desired quantile. Consider an array $x[1, 2, ..., n]$ of length n and we are interested in finding the w^{th} quantile. We assume that w is one of the indices of the array i.e. $w \in \{1, 2, \ldots, n\}$. Let q be the output of the partition step.

- If $q = w$, then we are done. The q-th element of the array i.e. $x[q]$ is the required w^{th} quantile.
- If $q < w$, then the required w^{th} quantile is greater than $x[q]$ and we have to call the recursion on the subarray $x[q + 1, ...n]$
- If $q > w$, then the required w^{th} quantile is less than $x[q]$ and we have to call the recursion on the subarray $x[1, 2, ..., q - 1]$

Pseudocode:

The algorithm is implemented by the function order_statistic which employs two helper functions PARTITION and FIND_ORDER_STAT.

The pseudocode of the algorithm is as follows:

```
PARTITION(x,p,r)

pivot = x[r]
i = p-1
for (j = p,p+1,...,r-1)
{
  if ( x[j] <= pivot)
  {
    i = i+1
    swap( x[i], x[j])
  }
}
swap(x[i+1], x[r])
return i+1

FIND_ORDER_STAT(x,w,p,r)

q = PARTITION(x,p,r)
if (q == w)
  return x[q]
elseif (q < w)
  FIND_ORDER_STAT(x,w,q+1,r)
else
  FIND_ORDER_STAT(x,w,p,q-1)

order_statistic(x,w)

n = size(x)
if(w <= n)
{
  FIND_ORDER_STAT(x,w,1,n)
}
```

Implementation of the algorithm in Rcpp:

Keeping in mind the differences in implementation of data structures in R and C++ , the implementation of the algorithm in Rcpp is as follows:

```
#include <Rcpp.h>

using namespace Rcpp;


// [[Rcpp::export]]

int PARTITION(NumericVector x, int p, int r){
  float temp, pivot;
  int i, j;
  pivot = x[r];
  i = p-1;
  for(j = p; j < r; j++){
    if(x[j] <= pivot){
      i = i+1;
      temp = x[j];
      x[j] = x[i];
      x[i] = temp;
    }
  }
  temp = x[i+1];
  x[i+1] = x[r];
  x[r] = temp;
  return i+1;
}

// [[Rcpp::export]]

float FIND_ORDER_STAT( NumericVector x, int w, int p, int r){
  int q;
  q = PARTITION(x, p, r);
  if (q < w){
    return FIND_ORDER_STAT(x, w, q+1, r);
  }
  else if (q > w) {
     return FIND_ORDER_STAT(x, w, p, q-1);
  }
  else if (q == w){
   return x[q];
  }
}

// [[Rcpp::export]]

float order_statistic(NumericVector x, int w){
  int n;
  n = x.size();
  if (w <= n){
    w = w-1;
```

```
    return FIND_ORDER_STAT(x, w, 0, n-1);
  }
}
```

Example:

```
x=sample(1:1000, 5000, replace = TRUE )
indices=sample(1:5000,10,replace=FALSE)
ord_stats_quicksort=rep(0,10)
for(i in 1:10)
{
  ord_stats_quicksort[i]=order_statistic(x, indices[i])
}
ord_stats_quicksort
```

```
##  [1] 121 212 259 319 758 925 162 252 316 519
```

```
sort(x)[indices]
```

```
##  [1] 121 212 259 319 758 925 162 252 316 519
```

```
identical(sort(x)[indices],as.integer(ord_stats_quicksort))
```

```
## [1] TRUE
```

From the example it is clear that `sort(x)[w]`=`order_statistic(x,w)`, as required. So, the program is working correctly (although correctness of the algorithm will not be proved here).

Empirical Study on Runtime:

For simulation based empirical runtime analysis, consider the following functions which will call `order_statistic(x,w)` for different choice of w, mainly $w = 1$ for minimum value i.e. $x_{(1)}$, $w = \lceil \frac{length(x)}{2} \rceil$ for median and $w = length(x)$ for maximum value i.e. $x_{(n)}$. Since we are interested in the asymptotic behaviour of the runtime, we consider input sizes greater than or equal to 10^4 and upto 10^7.

```
timefun = function(size, nrep , order.fun )
{
    x = replicate(nrep, runif(size), simplify = FALSE)
    system.time(lapply(x, order.fun))["elapsed"] / nrep
}

n = seq(10000, 10000000, by = 10000)

call_order_statistic_min = function(x){
  w = 1
  order_statistic(x, w)
}
```

```
call_order_statistic_med = function(x){
  w = ceiling(length(x)/2)
  order_statistic(x, w)
}

call_order_statistic_max = function(x){
  w = length(x)
  order_statistic(x, w)
}
```

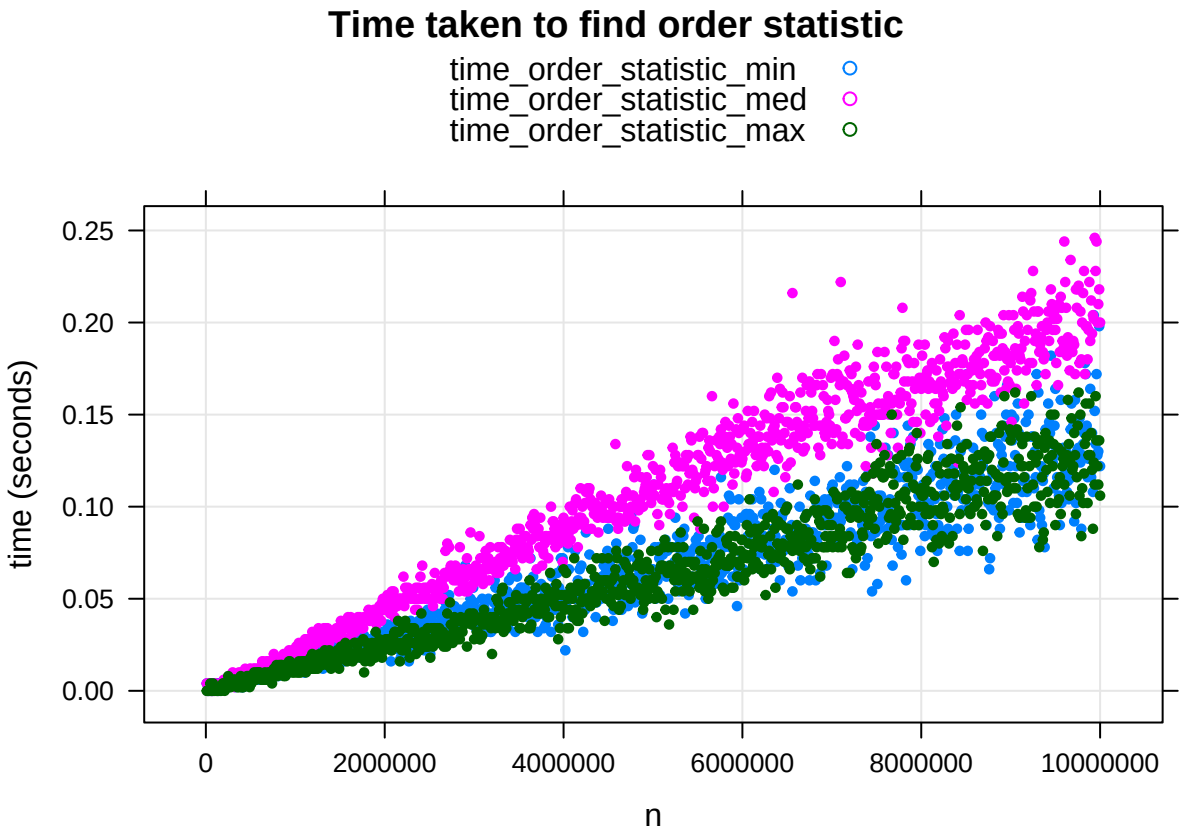
```
time_order_statistic_min = sapply(n, timefun, nrep = 5,
                                  order.fun = call_order_statistic_min)
time_order_statistic_med = sapply(n, timefun, nrep = 5,
                                  order.fun = call_order_statistic_med)
time_order_statistic_max = sapply(n, timefun, nrep = 5,
                                  order.fun = call_order_statistic_max)
```

Since the simulation took a long time to execute, we stored the simulated values in a csv file "Data on Empirical Runtimes.csv" and used the stored data to perform our analysis.

```
write.csv(data.frame(n,time_order_statistic_min,time_order_statistic_med,time_order_statistic_max),
file="Data on Empirical Runtimes.csv")
```

Plot of time taken to find specific order statistic for varying length of array is shown below:

```
data=read.csv("Data on Empirical Runtimes.csv")
n=data[,2]
time_order_statistic_min=data[,3]
time_order_statistic_med=data[,4]
time_order_statistic_max=data[,5]
library(lattice)
xyplot( time_order_statistic_min + time_order_statistic_med + time_order_statistic_max ~ n,
        type = "p" ,auto.key = list(space = "top"),grid = TRUE,
        ylab = "time (seconds)", main = "Time taken to find order statistic", pch = 20)
```



From the above plot for time taken to find minimum , median and maximum, it is clear that average time is almost same for all these cases, although time taken to find the median value is slightly higher than others. Summary of the runtimes is as follows:

```
summary(time_order_statistic_min)
```

```
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.03400 0.06400 0.06712 0.09650 0.20400
```

```
summary(time_order_statistic_med)
```

```
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.0000  0.0560  0.1080  0.1074  0.1600  0.2460
```

```
summary(time_order_statistic_max)
```

```
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00000 0.02800 0.06000 0.06344 0.09600 0.16200
```

We now try to determine the exact order of growth of the average runtime is n or $n\,log\,n$.

Given a function $g : \mathbb{N} \to \mathbb{R}$, define

$$\Theta(g(n)) = \{f(n) \quad | \quad \exists\ c_1, c_2 > 0 \text{ and } N \in \mathbb{N} \text{ such that } n \geq N \implies 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

That is , $f(n) \in \Theta(g(n))$ if $f(n)$ can be asymptotically bounded on both sides by multiples of $g(n)$.

In order to check asymptotic bound of $f(n)$ (which is time taken by `order_statistic(x,w)` here) we will consider the ratio $\frac{f(n)}{g(n)}$. As time taken by the function is almost same irrespective of any value of w (for minimum , maximum, median) , we restrict ourselves in $w = \lceil \frac{length(x)}{2} \rceil$. As per the hypothesis is concerned , we have two different choices of $g(n)$ one is $g(n) = n$ and another is $g(n) = nlog(n)$

The ratio $\frac{f(n)}{g(n)}$ may have three types of limiting value,

$$\text{If } \frac{f(n)}{g(n)} = 0 \text{ , for large values of } n \text{ , then order of } f(n) \text{ is lower than that of } g(n) \tag{4}$$

$$\text{If } \frac{f(n)}{g(n)} = \infty \text{ , for large values of } n \text{ , then order of } f(n) \text{ is higher than that of } g(n) \tag{5}$$

$$\text{If } c_1 \leq \frac{f(n)}{g(n)} \leq c_2 \text{ , for large values of } n \text{ with } 0 < c_1 < c_2 < \infty \text{ , then order of } f(n) = \Theta(g(n)) \tag{6}$$

We want to determine whether the average runtime is $\Theta(n)$ or $\Theta(n \log n)$. Since the scatterplot of the average runtime against n and $n \log n$ is almost linear, we fit two separate simple linear regression models of the average runtime on n and $n \log n$, and perform a hypothesis test to determine which one fits the data better. For performing the hypothesis test we propose the following test based on bootstrapping:

- We want to test whether the difference between the adjusted R^2 values for the two models fitted is significant or not.
- We test H_0 : The difference is not significant against H_1 : The model based on n is better.
- Fit a linear model of the average runtime (say y) on n and $n \log n$, separately.
- Compute the adjusted R^2 for each model.
- Compute 1000 bootstrap samples each of size 100 from the data.
- Compute the R^2 values for the linear models of y on n and y on $n \log n$ fitted on each bootstrap sample, separately.
- Compute the differences between the corresponding R^2 values
- Compute the appropriate one sided empirical p-value under the null hypothesis and conclude.

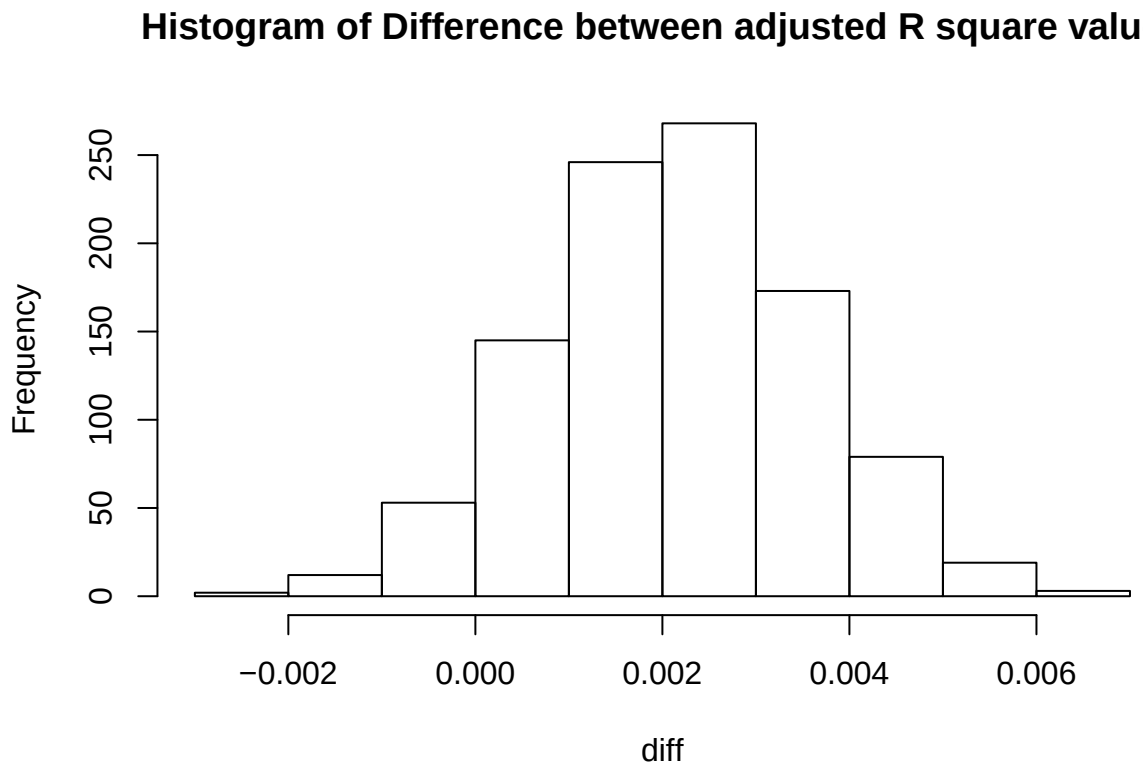
To compute the empirical p-value and compute the bootstrap replications we define the following functions and perform the following computations:

```
adjrsq=function(data){
  summary(lm(data[,2]~data[,1]))$adj.r.squared
}
nlogn=n*log(n)
Rsq_n=adjrsq(data.frame(n=n,time=time_order_statistic_med))
Rsq_nlogn=adjrsq(data.frame(n=n*log(n),time=time_order_statistic_med))

simulate=function(data,size){
  data[sample(nrow(data),size,replace=T),]
}

bootadjrsq = function(data , bootsize, nboot)
{
    x = replicate(nboot, simulate(data,bootsize), simplify = FALSE)
    sapply(x,adjrsq)
}
bootRsq_n=bootadjrsq(data.frame(n=n,time=time_order_statistic_med),200,100000)
```

```
bootRsq_nlogn=bootadjrsq(data.frame(n=n*log(n),time=time_order_statistic_med),200,100000)
write.csv(data.frame(Rsq_n=bootRsq_n,Rsq_nlogn=bootRsq_nlogn),
file="Bootstrap distribution of adjusted Rsquare.csv")
```

Since the simulation took a long time to execute, we stored the simulated data in a csv file named "Bootstrap distribution of adjusted Rsquare.csv" and used the stored data to perform our analysis.

```
bootdata=read.csv("Bootstrap distribution of adjusted Rsquare.csv")
diff=bootdata$Rsq_n-bootdata$Rsq_nlogn
hist(diff,main="Histogram of Difference between adjusted R square values")
```




```
summary(diff)
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -0.002567  0.001152  0.002149  0.002139  0.003080  0.006747
```

```
pvalue=sum(as.numeric((diff<=0)))/length(diff)
pvalue
```

```
## [1] 0.067
```

Thus we see that the one sided empirical p-value, representing an approximation to the probability that the R^2 value corresponding to the regression of y on n is smaller than the R^2 value corresponding to the regression of y on $n\,log\,n$, is significantly small at say, 10 percent level of significance. Hence we conclude that the average case runtime of `order_statistic(x,w)` is $\Theta(n)$.

Empirical Study on Variance of Runtime:

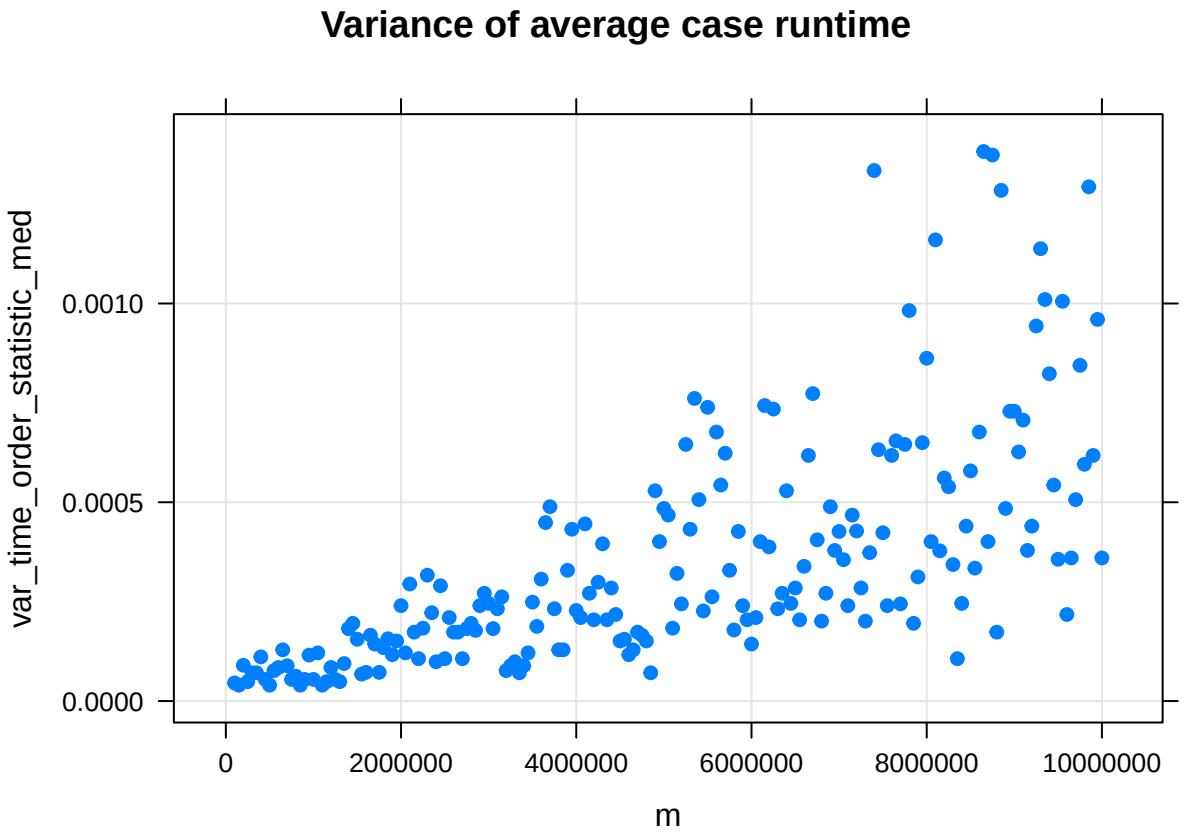
In this section we denote **m** as the length of an array. To carry out an empirical study to find the order of growth of variance of average runtime of `order_statistic(x,w)`, we will run the program 10 times for fixed input array size and compute the sample variance of these 10 time values and try to initialize the same idea like above. That is we will study the ratio $\frac{f(m)}{g(m)}$, with variability of time to get the order statistic for fixed m as $f(m)$.

```
timevarfun = function(size, nrep , order.fun) {
  t = vector(mode = "numeric")
  for ( i  in 1 : nrep) {
    x = runif(size)
    t = c(t , system.time(order.fun(x))["elapsed"])
  }
  var(t)
}
m = seq(100000, 10000000, by = 50000)
var_time_order_statistic_med = sapply(m, timevarfun, nrep = 10,
                                      order.fun = call_order_statistic_med)
write.csv(data.frame(m,var_time_order_statistic_med),
file="Variance of Empirical Runtimes.csv")
```

Since the simulation took a long time to execute, we stored the simulated values in a csv file "Variance of Empirical Runtimes.csv" and used the stored data to perform our analysis.

Let us plot these variances against the length of the array.

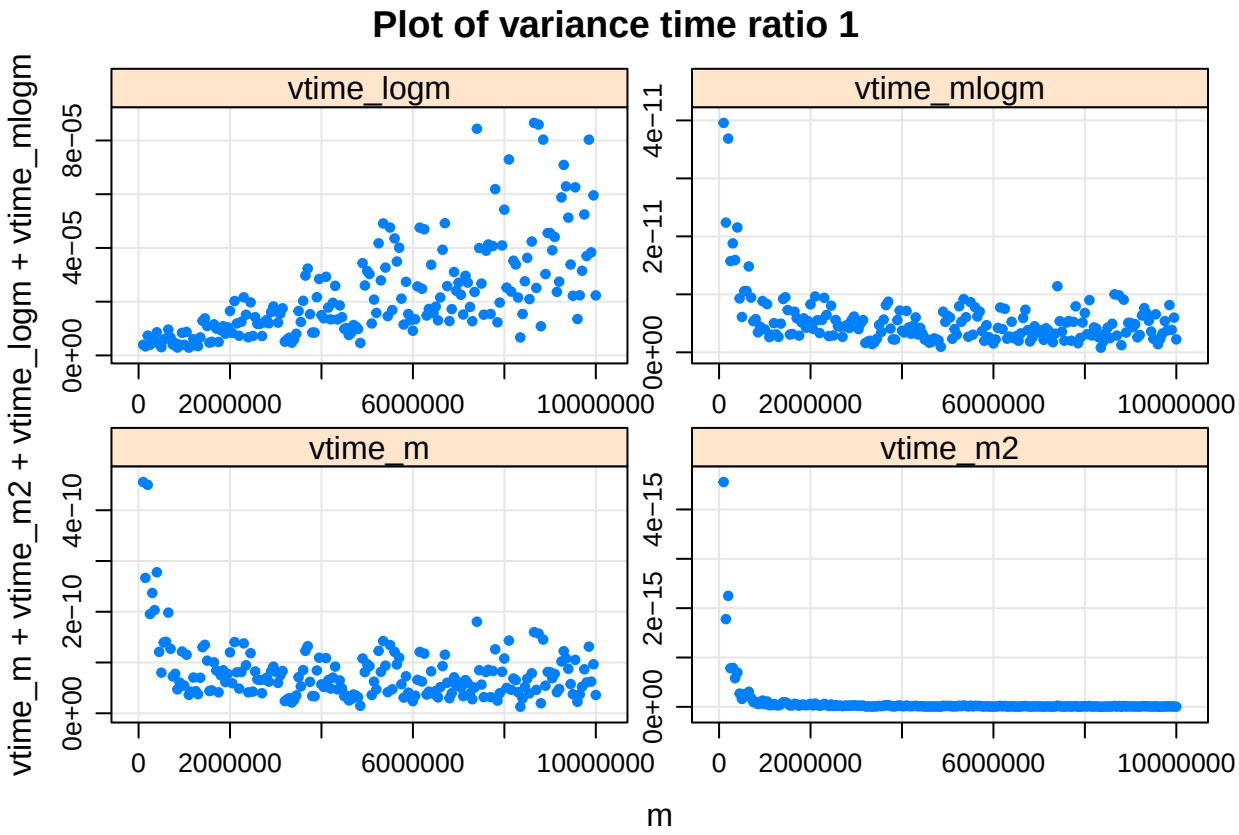
```
data=read.csv("Variance of Empirical Runtimes.csv")
m=data[,2]
var_time_order_statistic_med=data[,3]
library(lattice)
xyplot(var_time_order_statistic_med ~ m, grid = TRUE, jitter = TRUE,
       main = "Variance of average case runtime", pch = 19)
```



From the above scatterplot, we can find that there is a positive correlation between variance of runtime and length of the array. We can think of variance of runtime as an increasing function of m (length of the array) with order $g(m)$, and m,m^2 and $mlog(m)$ as possible choices of $g(m)$. We will again study the ratio $\frac{f(m)}{g(m)}$ with above $g(m)$.

```
vtime_m = var_time_order_statistic_med/m
vtime_m2 = var_time_order_statistic_med/(m^2)
vtime_logm = var_time_order_statistic_med/log(m)
vtime_mlogm = var_time_order_statistic_med/(m*log(m))
```

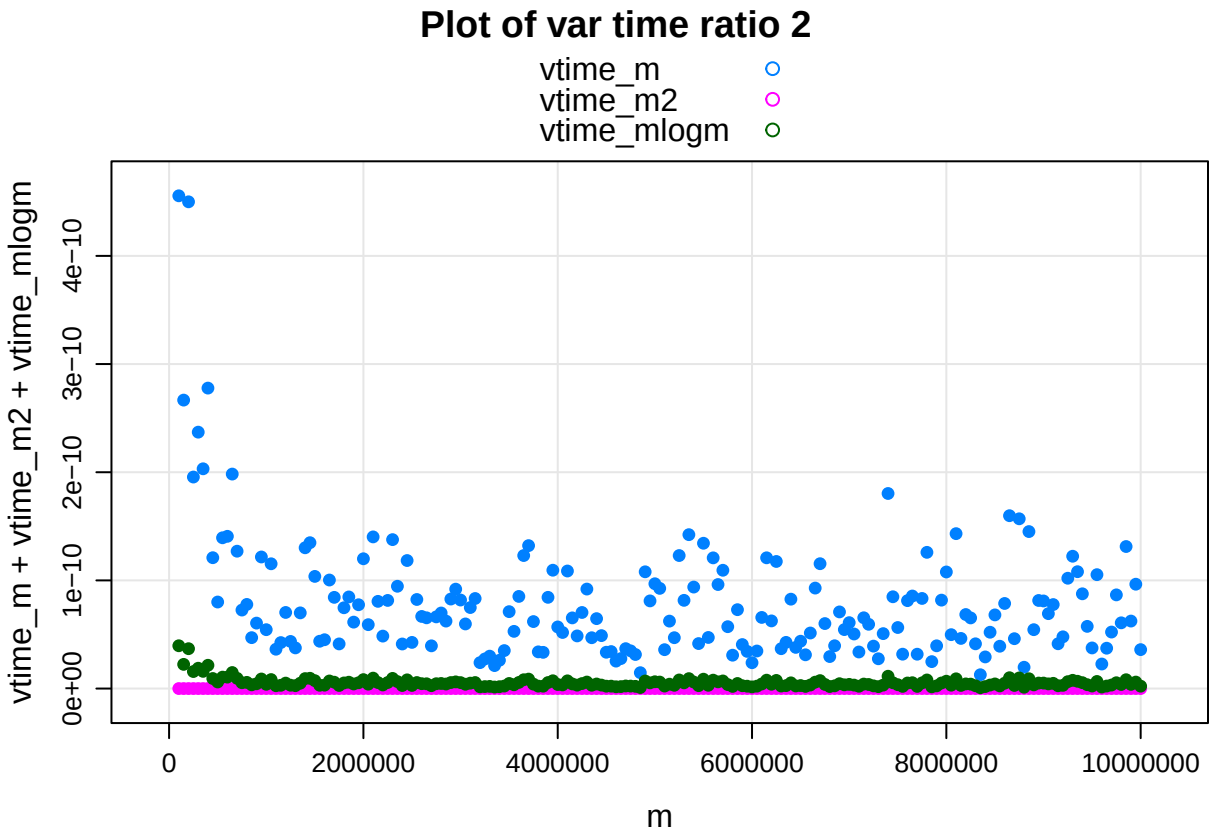
```
xyplot(vtime_m + vtime_m2 +vtime_logm + vtime_mlogm ~ m,
       outer = TRUE, scale = list(x = "free", y = "free"), auto.key = list(space = "right")
       , main = "Plot of variance time ratio 1", grid = TRUE, pch = 20)
```



From the diagram 'Plot of variance time ratio 1' the following can be concluded :

- $\frac{f(m)}{g(m)}$ is not constant in m for $g(m) = log(m)$ rather this ratio is an increasing function of m (case (5)) . So $f(m) \notin \Theta(log(m))$
- Limiting value of $\frac{f(m)}{g(m)}$ is constant for m , m^2 , $mlog(m)$ as a choice of $g(m)$.

```
xyplot( vtime_m + vtime_m2 + vtime_mlogm  ~ m,
        scale = list(x = "free", y = "free"), auto.key = list(space = "top"),
        main ="Plot of var time ratio 2", grid = TRUE, pch = 16)
```



From this plot we see that, the limiting value of the ratio $\frac{f(m)}{g(m)}$ is very close to zero (Pink points in 'Plot of var time ratio 2'), so $f(m)$ has lower order of growth than that of m^2 (like case (4)). For $g(m) = m$ or $m\,log(m)$, this fraction converges to a positive real, which is clear from 'Plot of var time ratio 1' (like case (6)). But the points for $g(m) = m$ (Blue points in 'Plot of var time ratio 2') is more dispersed (band of points is much wider) than the points for $g(m) = m\,log(m)$ (Green points in 'Plot of var time ratio 2').

So, through this simulation study we can conclude that variance of the average case runtime of `order_statistic(x,w)` is $\Theta(m\,log\,(m))$.