Timing sort functions With Python

Taylor Blair

10/27/2020

Import libraries

This homework assignment utilises two programming languages, R and Python.

R is necessary as the PDF is created using R markdown. In order to run python in R, a wrapper is being utilised. If you are interested in seeing the difference in performancesm please skip to the section

Importing R libraries

```
library(reticulate)
matplotlib <- import("matplotlib")
matplotlib$use("Agg", force = TRUE)
library(ggplot2)
```

Importing Python Libraries

```
import matplotlib.pyplot as plt
import numpy as np
import math
import time
import pandas as pd
import random
```

1. What the homework asked for

There are a number of sorting functions often used for sorting. Here were the four you wanted timed.

Helper Functions

```
new_list = lambda n :random.sample(list(range(0,n)), n)
#Creates a randomly shuffled list of n length

def check_sort(lst):
  for y in range(0,len(lst)-1):
    if lst[y]>lst[y+1]:
      return False
  return True
# Checks if a list is sorted

def time_sort_funcs(i, sort_elements, sort_func):
```

```
  sort_times = []
  for x in range(0,i):
    list_to_sort = new_list(sort_elements)
    start = time.perf_counter()
    sort_func(list_to_sort)
    end = time.perf_counter()
    sort_times.append(end-start)
  return sort_times
```

Quicksort

```
def partition ( ls , left , right ):
  pivot = random . randint ( left , right )
  ls [ pivot ] , ls [ left ] = ls [ left ] , ls [ pivot ]
  less = left + 1
  greater = right
  while less <= greater :
    if ls [ less ] < ls [ left ]:
      less = less + 1
    else :
      ls [ less ] , ls [ greater ]= ls [ greater ] , ls [ less ]
      greater = greater - 1
    ls [ left ] , ls [ less - 1] = ls [ less - 1] , ls [ left ]
  return less - 1
def qshelp ( ls , first , last ):
  if first < last :
    pivot = partition ( ls , first , last )
    qshelp ( ls , first , pivot -1)
    qshelp ( ls , pivot +1 , last )

def quicksort ( ls ):
  qshelp ( ls , 0 , len ( ls ) -1)
```

Bubble sort

```
def bubble(lst):
  i = 0
  while check_sort(lst)==False:
    if lst[i%len(lst)]>lst[(i+1)%len(lst)] and (i+1)%len(lst)!=0:
      lst[i%len(lst)], lst[(i+1)%len(lst)] = lst[(i+1)%len(lst)], lst[i%len(lst)]
    print(lst)
    i+=1
```

Insertion sort

```
def insert_sort(lst):
  sorted = [lst[0]]
  for x in lst[1:]:
```

```
    sorted.insert(len([1 for y in sorted if x>y]) , x)
  lst[:] = sorted
```

Merge sort

```
def merge(a,b):
  c = []
  i, j = 0,0
  while i+j < len(a)+len(b):
    if i>= len(a) or (j<len(b) and b[j] <=a[i]):
      c.append(b[j])
      j+=1
    elif j>=len(b) or a[i]<= b[j]:
      c.append(a[i])
      i+=1
  return c

def mergesort(aList):
  if len(aList) <=1:
    return aList
  else:
      mid = len(aList)//2
      return merge(mergesort(aList[:mid]), mergesort(aList[:mid]))
```

1B. Creating Runtimes

I set the number of iterations to $10,000$, and the number of elements to sort to 30. This means there are 30! possible orders

```
sort_funcs = [bubble, insert_sort, quicksort, mergesort]
sort_func_names = ["Bubble", "Insertion", "Quick", "Merge"]

sort_runtimes = pd.DataFrame(columns = sort_func_names)
iters = 10000
i = 30


for x in range (0,len(sort_funcs)):
  sort_runtimes[sort_func_names[x]] = time_sort_funcs(iters, i, sort_funcs[x])
```

```
sort_runtimes = sort_runtimes.reindex(sort_runtimes.mean().sort_values(ascending=False).index, axis=1)
```

2. Visualizing results

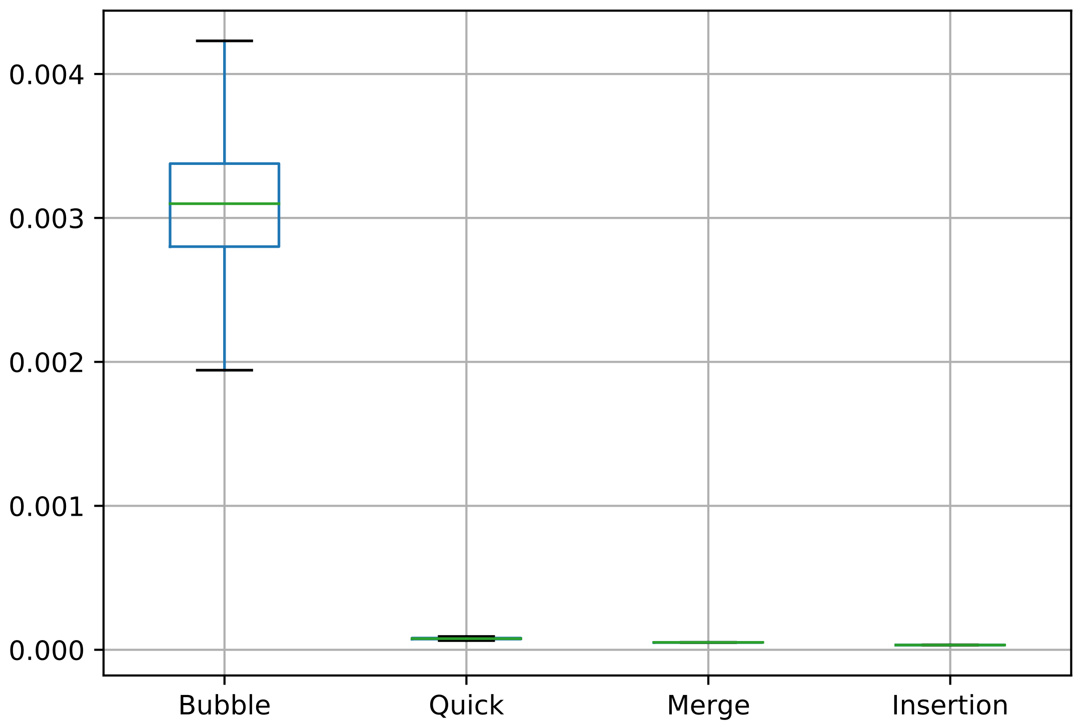
A brief statistical descrption of each column

```
sort_runtimes.describe()
```

```
##               Bubble         Quick         Merge     Insertion
```

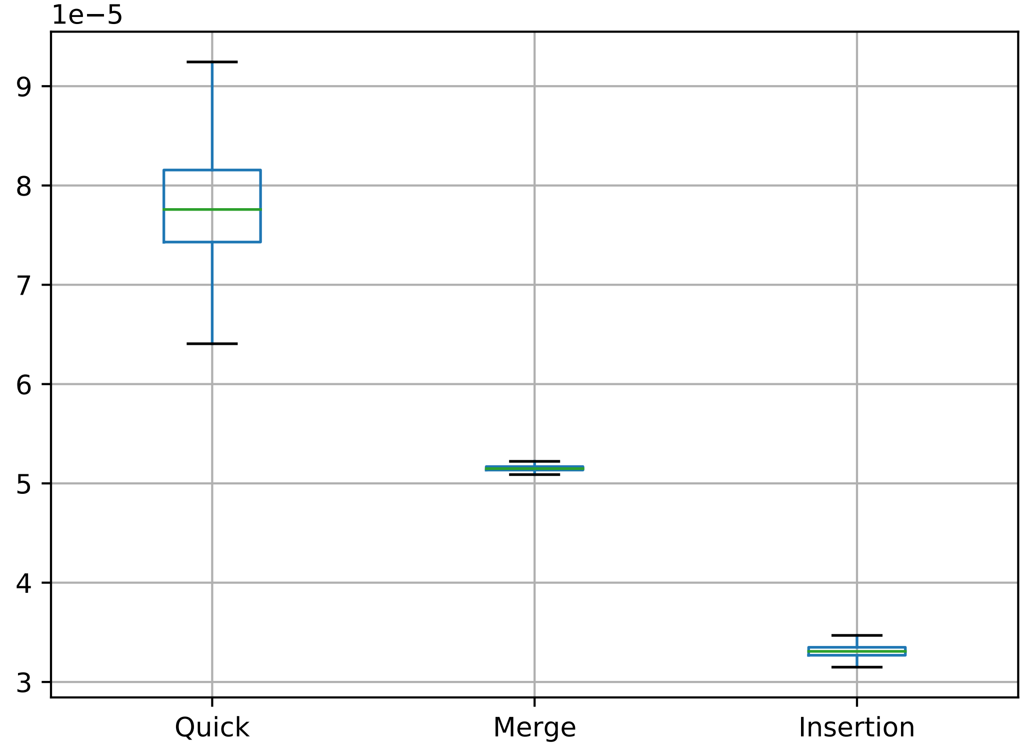
```
## count  10000.000000  10000.000000  10000.000000  10000.000000
## mean       0.003174      0.000079      0.000053      0.000034
## std        0.000804      0.000011      0.000008      0.000005
## min        0.001258      0.000064      0.000051      0.000031
## 25%        0.002801      0.000074      0.000051      0.000033
## 50%        0.003099      0.000078      0.000051      0.000033
## 75%        0.003378      0.000082      0.000052      0.000033
## max        0.013781      0.000271      0.000316      0.000169
```

```
sort_runtimes.boxplot(showfliers=False)
```



Without Bubble Sort

```
sort_runtimes.drop('Bubble',1).boxplot(showfliers=False)
```



3. Additional Python Sorting Functions

```
def selectionsort(lst):
  for x in range(0, len(lst)):
    swap_on = lst[x:].index(min(lst[x:]))
    lst[swap_on+x], lst[x] = lst[x], lst[swap_on+x]
```

Countsort, also reffered to as bea and gravity sort

```
def countsort(lst):
  ident_lst= [0]*len(lst)
  for x in lst:
    ident_lst[x] +=1
  lst[:] = [y for y in range (0,len(ident_lst)) for z in range(0,ident_lst[y])]
```

Heap sort, I did not make this function. I

```
def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
```

```
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]  # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    # Since last parent will be at ((n//2)-1) we can start at that location.
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # swap
        heapify(arr, i, 0)
```

Pythons built in sort method is Timsort

```
def timsort(lst):
  lst.sort()
```

```
sort_funcs = [selectionsort, countsort, heapSort, timsort]
sort_func_names = ["Selection", "Count", "Heap", "Tim"]

for x in range (0,len(sort_funcs)):
  sort_runtimes[sort_func_names[x]] = time_sort_funcs(iters, i, sort_funcs[x])


sort_runtimes = sort_runtimes.reindex(sort_runtimes.mean().sort_values(ascending=False).index, axis=1)
```

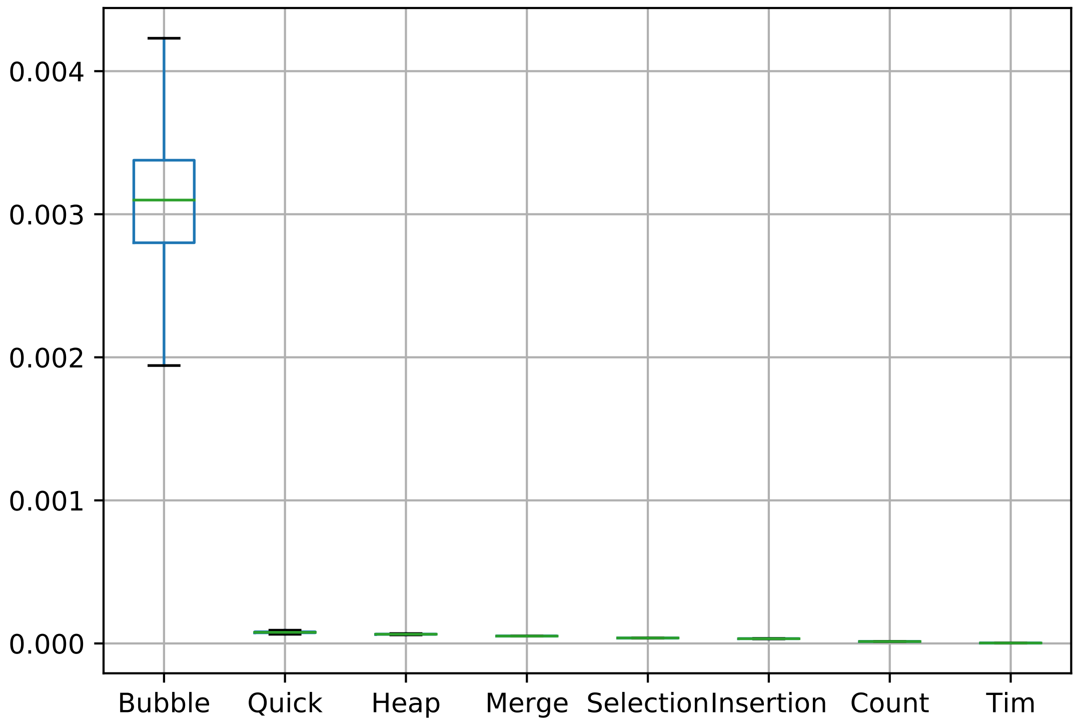
```
df_drop_old = sort_runtimes.drop(["Bubble", "Insertion", "Quick", "Merge"], 1)

df_drop_old.describe()
```

```
##                 Heap     Selection         Count           Tim
## count  10000.000000  10000.000000  10000.000000  1.000000e+04
## mean       0.000067      0.000039      0.000014  2.993833e-06
## std        0.000012      0.000004      0.000002  5.464354e-07
## min        0.000057      0.000036      0.000013  2.661720e-06
## 25%        0.000063      0.000038      0.000013  2.906658e-06
## 50%        0.000064      0.000038      0.000013  2.965331e-06
## 75%        0.000066      0.000038      0.000014  3.028661e-06
```

```
## max        0.000466      0.000099      0.000216  4.326645e-05
```

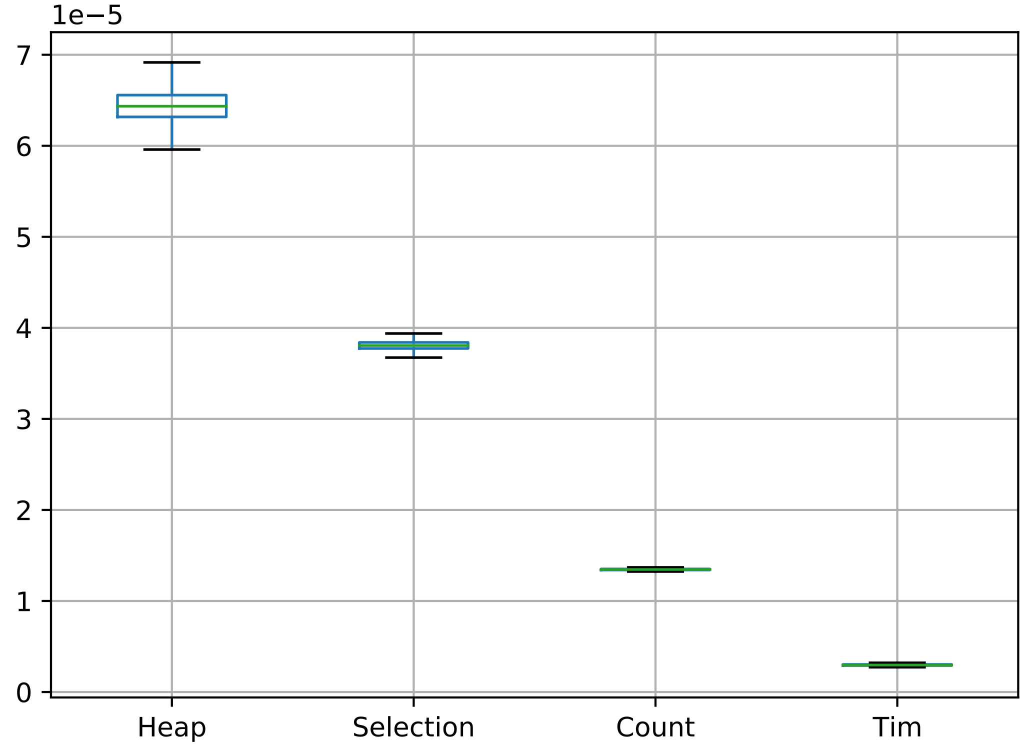
```
sort_runtimes.boxplot(showfliers=False)
```



Boxplot

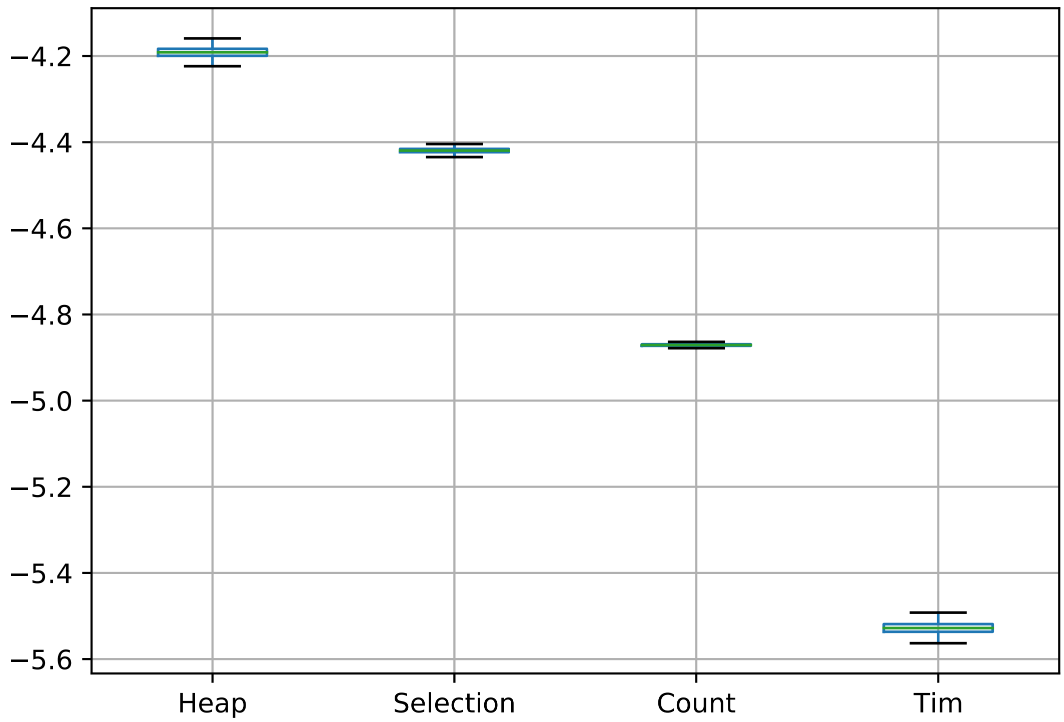
Outlier points

```
df_drop_old.boxplot(showfliers=False)
```



Log Scale Boxplot

```
np.log10(df_drop_old).boxplot(showfliers=False)
```



5. Performance of R vs Python in Rstudio

Note: There is currently a preview build of R studio that allows for python to be utilised without a wrapper, but I did not have the time to debug issues that could disrupt my workflow in stats and CS.

To create the PDF you are hopefully grading, I used Rstudio which allows users to "knit" a document (analgous to building a LaTeX PDF).

To test if there is a significant difference between R and Python performance, I created several computational expensive algorithims and compared their runtimes.

Prime Testing with Wilsons Theorm

Wilsons theorm is a easy to ride, yet inefficient solution to finding primes. If $(n-1)! = -1 \mod n$ then a number is prime. This formula can also be rewritten as $(n-1)! \mod n = n-1$ or $(n-1)! \mod n+1 = n$

While the formula is simple to write, the factorial and modulo make it computationally expensive (*as a result it is not used to check or prove primes*). In addition, the two functions are standardized across coding languages, so we expect them to have similar complexity.

Python version of Wilsons Theorm

There is no built in factorial equation, so I used the math package

```
wilson_py = lambda x : math.factorial(x-1)%x==x-1
```

R version of Wilsons Theorm

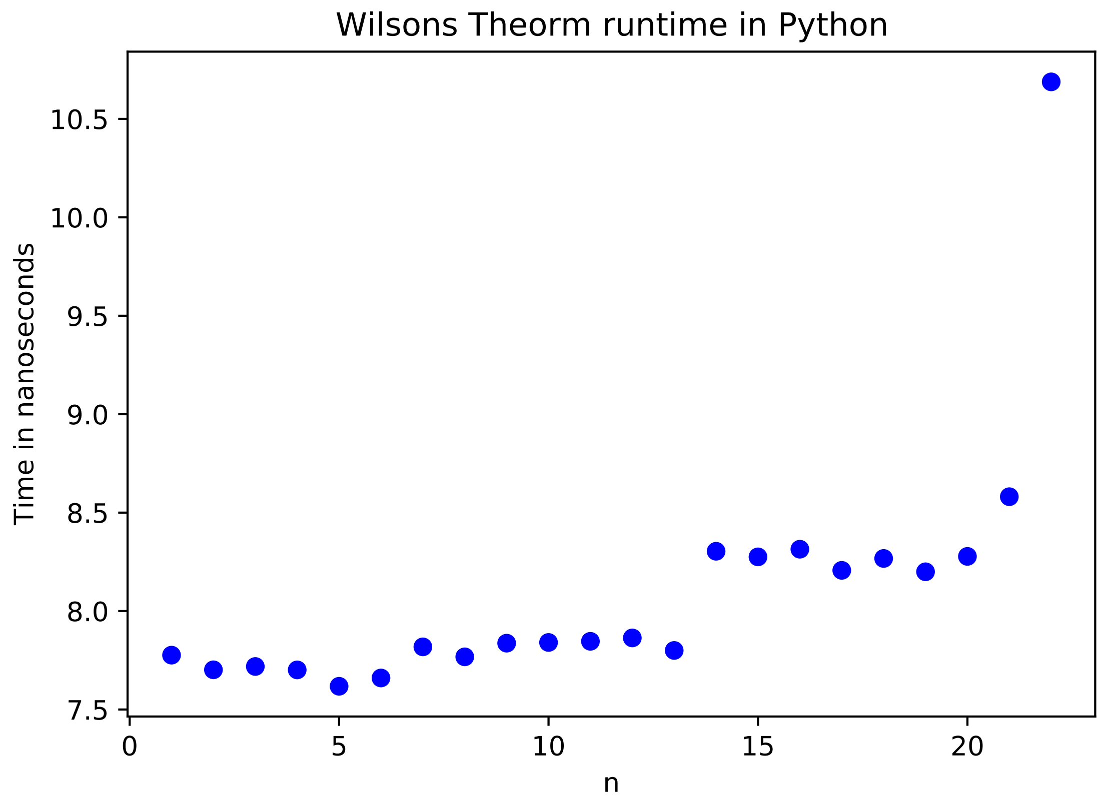
R has a built in method for factorial, but it is limited. The function additionally cannot compute `wilson_r(23)` or any $n > 22$. As R has a 64 bit limit for integers and $log_2(23!) > 64$.

```
wilson_r <- (function(x) factorial(x-1)%%x==x-1)
```

Timing Python runtime

```
py_wilson_runtime = []
for x in range(1,23):
  holder_times = []
  for y in range(0,10000):
    py_start_time = time.perf_counter()
    wilson_py(x)
    py_end_time = time.perf_counter()
    holder_times.append(py_end_time-py_start_time)
  py_wilson_runtime.append(np.mean(holder_times)*(10**7))
```

```
plt.plot(list(range(1,23)), py_wilson_runtime, "bo")
plt.title("Wilsons Theorm runtime in Python")
plt.xlabel("n")
plt.ylabel("Time in nanoseconds")
plt.show()
```



Timing R

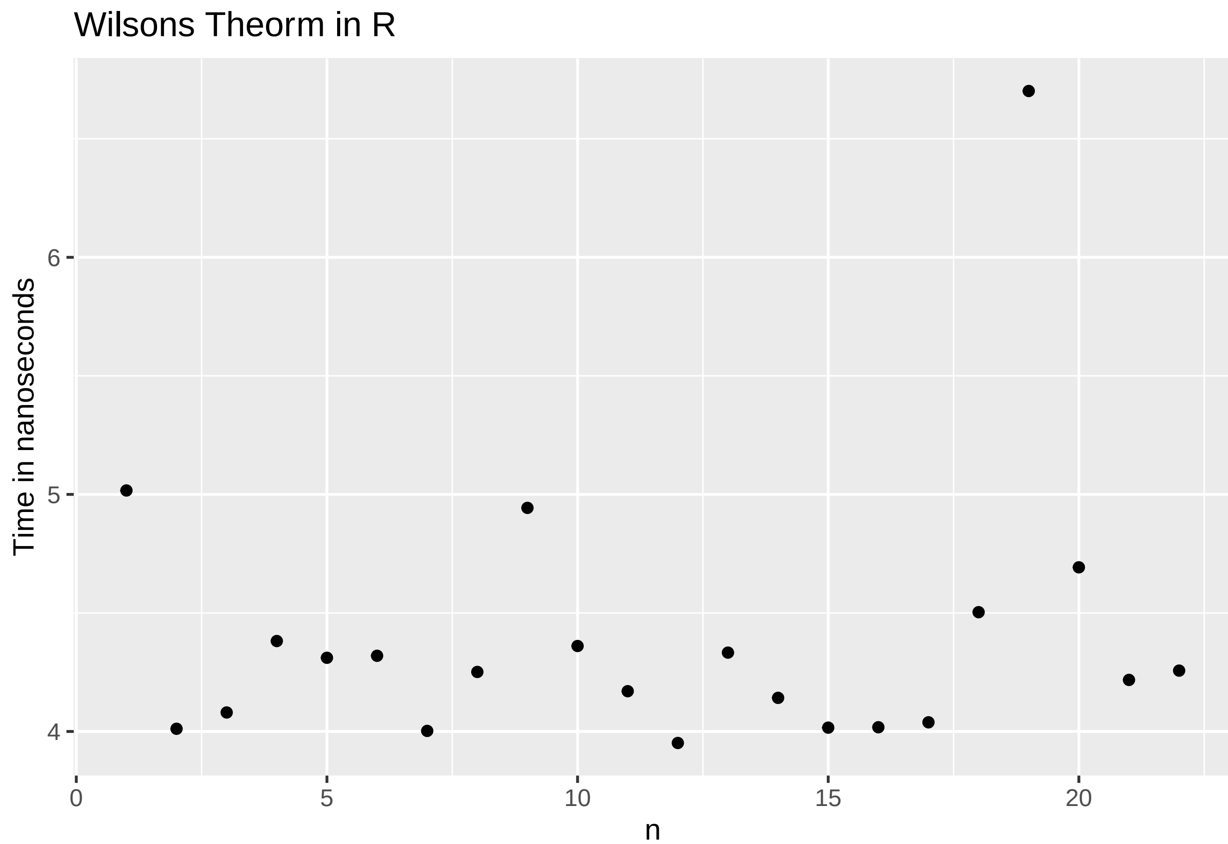
Indices in R start at 1, so `c(1:10)` in R is equvilent to `list(range(1,11))` in Python.

```
r_wilson_runtime <- c(1:22)

for (x in c(1:22)){
  holder_times <- c(1:10000)
  for (y in c(1:10000)){
    start_time <- as.numeric(Sys.time())
    wilson_r(x)
    end_time <- as.numeric(Sys.time())
    holder_times[y]<- end_time-start_time
  }
  r_wilson_runtime[x] <-mean(holder_times*(10**6))
}
```

```
wilson_df <- data.frame(n=c(1:22), n_x=r_wilson_runtime)

ggplot(wilson_df, mapping = aes(n, n_x)) +
  geom_point() +
  labs(title = "Wilsons Theorm in R", y= "Time in nanoseconds")
```



6. Unfinished work

There was going to be additional work compare R, Python, and other language built in sort methods. I was also going to make a tile graph that showed fastest runtimes. That didn't happen

But due to unforseen circumstances, I needed to change my attention. Regardless, I hope you enjoyed this brief analysis into a few python sort function.

And because David asked for it. Here's bogo sort as a histogram. Only 6 elements as that would be a long loooong runtime.

```
def bogosort(lst):
    while not check_sort(lst):
        random.shuffle(lst)
    return lst

sort_runtimes["Bogo 6"] = time_sort_funcs(iters, 6, bogosort)
```

```
sort_runtimes[["Bogo 6", "Bubble"]].describe()
```

```
##              Bogo 6        Bubble
## count  10000.000000  10000.000000
## mean       0.003840      0.003174
## std        0.003851      0.000804
## min        0.000001      0.001258
## 25%        0.001131      0.002801
## 50%        0.002627      0.003099
```

```
## 75%         0.005281       0.003378
## max         0.036541       0.013781
```

```
sort_runtimes[["Bogo 6", "Bubble"]].boxplot()
```