170025298 W10-Complexity Tutor: Michael Weir 20/Nov./2018

Outline:

1 Coding Analyse
1.1 Design
1.1.1 QuickSort:
Quick Sort is designed as an recursive function that follows the following algorithm:
Define bigger, smaller and equal list and use for loop to add elements bigger/ smaller/ equal elements to reflected list. Then when bigger and smaller is empty which shows that there are only one variable in the list and this will return equal list; when they are not empty, which will leads to recursion until it will be divided into the smallest element with only one variable. Sorted will add elements from small to big and will finally return the whole list.
Exceptions are prevented when only one element inside of the list or list is null.

1.1.2 EditDistance:
EditDistance returns a sorted list and have a random swap method which increase the complexity.

1.1.3 OutputData:
OutputData is used to output the data of complexity and time spends on running the program. It uses print writer to out put the data into a csv file.

1.2 Test:
1.2.0 Random Number Class: This class create a series of random number list split with “,” which uses Math.random method. The series will be converted to the unsorted list in the test class.

1.2.1 Random Number Test: This test aims at testing the random number has been parsed to integer and saved to the array list.

1.2.2 nullTest & none element test: This tests when input list is null or input no element to the sort class to prove that the error will not happens to the program.

1.2.3 QuickSort Test: This test will test quick sort has successfully sorted the list and it will print the time-spent on running the program.

1.2.4 Special Test Class: This test tests the sorting method applicates with huge size of list and ways of testing is similar to the previous tests.

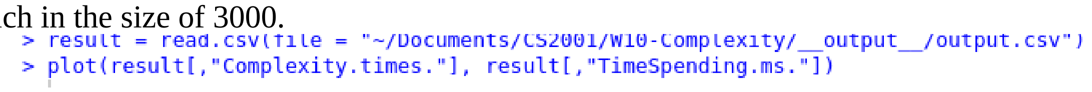
Hint: Tests are similar when testing merge sort and bubble sort.

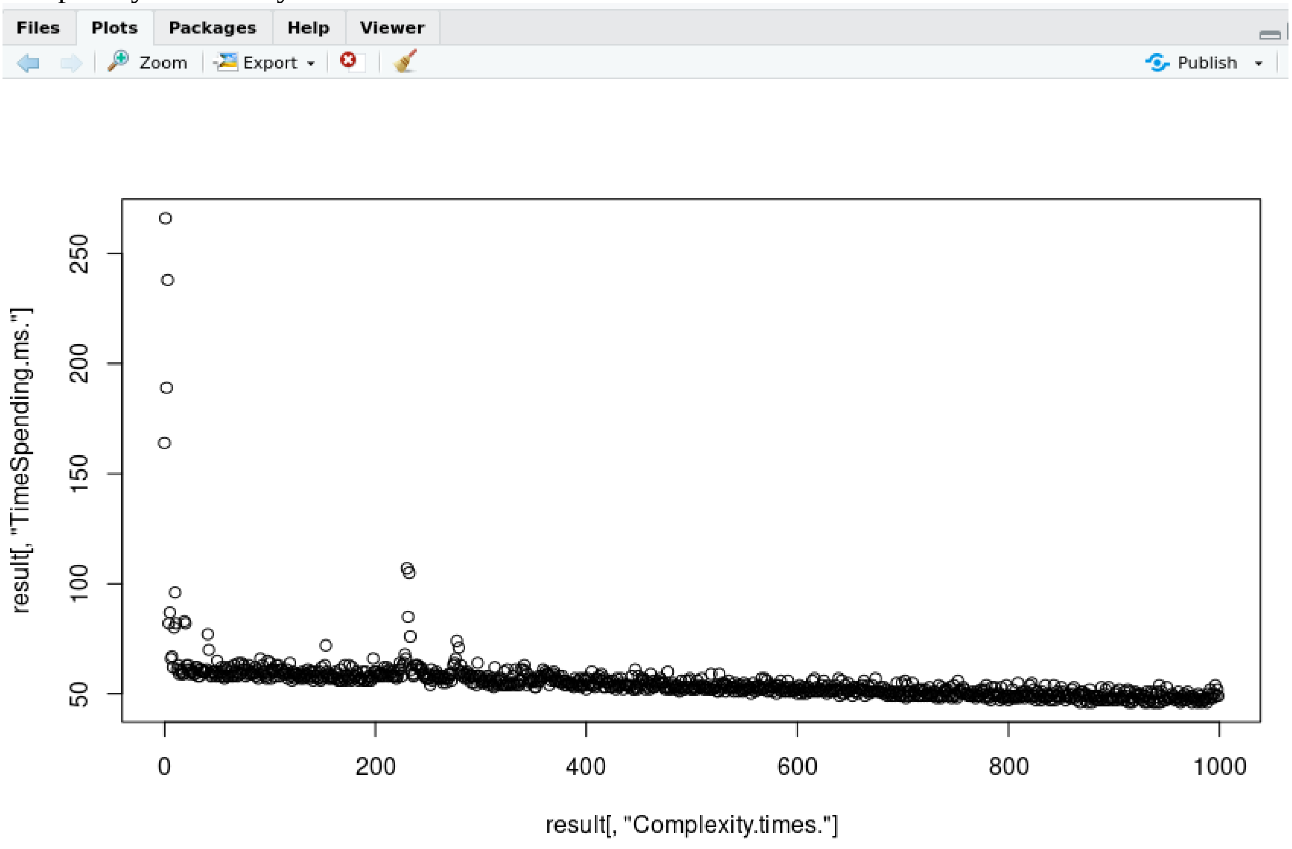
1.3 Extension:
1.3.1 Bubble Sort:
Bubble sort will go through the list twice and when current element is bigger than the next element, the bubble sort will swap the element with next element.

1.3.2 Selection Sort:
Selection sort will insert the minimum value into the sorted list and return the full list of the list when the recursive function returns empty list.

1.3.3 Merge Sort:
Merge sort was learned from github and related description, reference and leaning process have been noted inside of the program. This method only operate within the list using different index, the merge sort divide the index instead of the list and merge the reflected element of the index at the end.

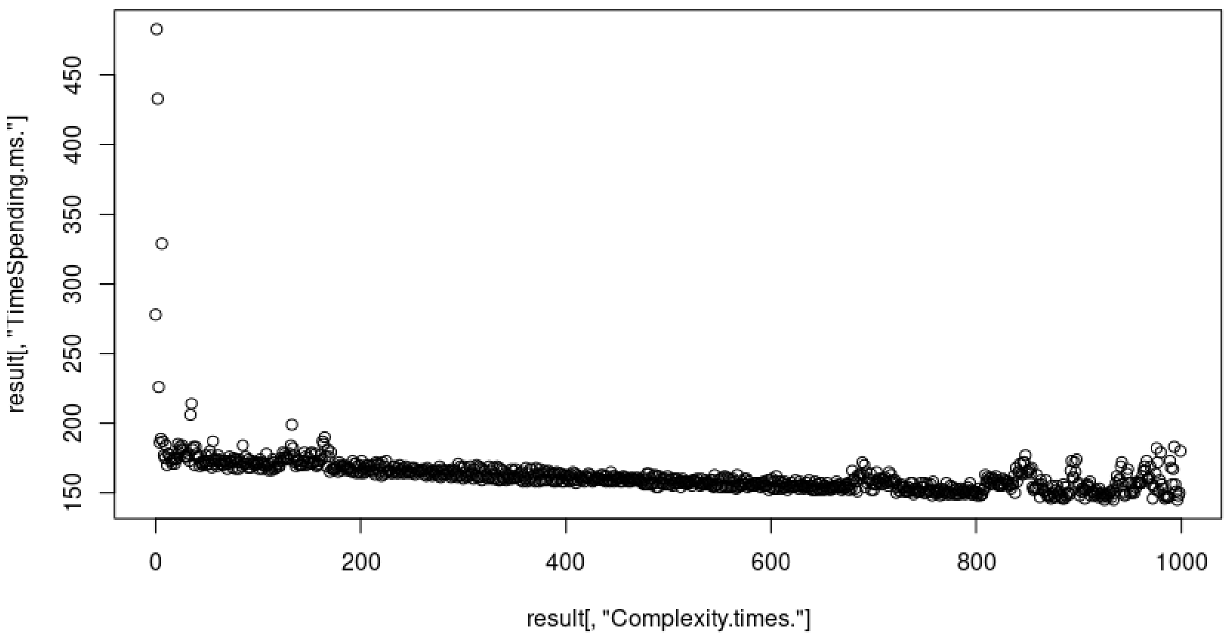
1.3.2

2.1 Quick sort
In order to analyse the data, java created a csv file which includes using quick sort to sort the data which in the size of 3000.

```
> result = read.csv(file = "~/Documents/CS2001/W10-Complexity/__output__/output.csv")
> plot(result[,"Complexity.times."], result[,"TimeSpending.ms."])
```

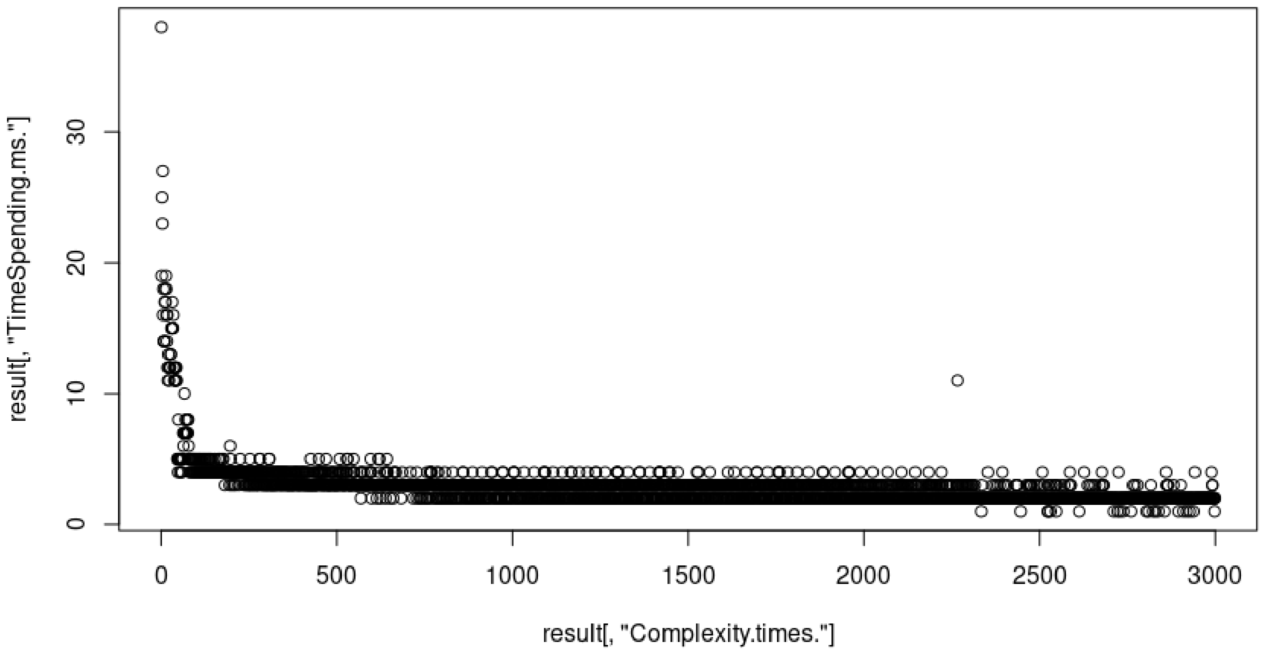
Data analyse using Rstudio and a graph was drawn using the command above.
To prevent systematic error and I will control variables of size of the list and control the variable of complexity and finally I will draw the final conclusion.


Even though with the increasing of the complexity, the list goes from sorted to unsorted, however the result is tricky that the result against what I have thought previously that as the complexity goes up, the time spending comes lower.

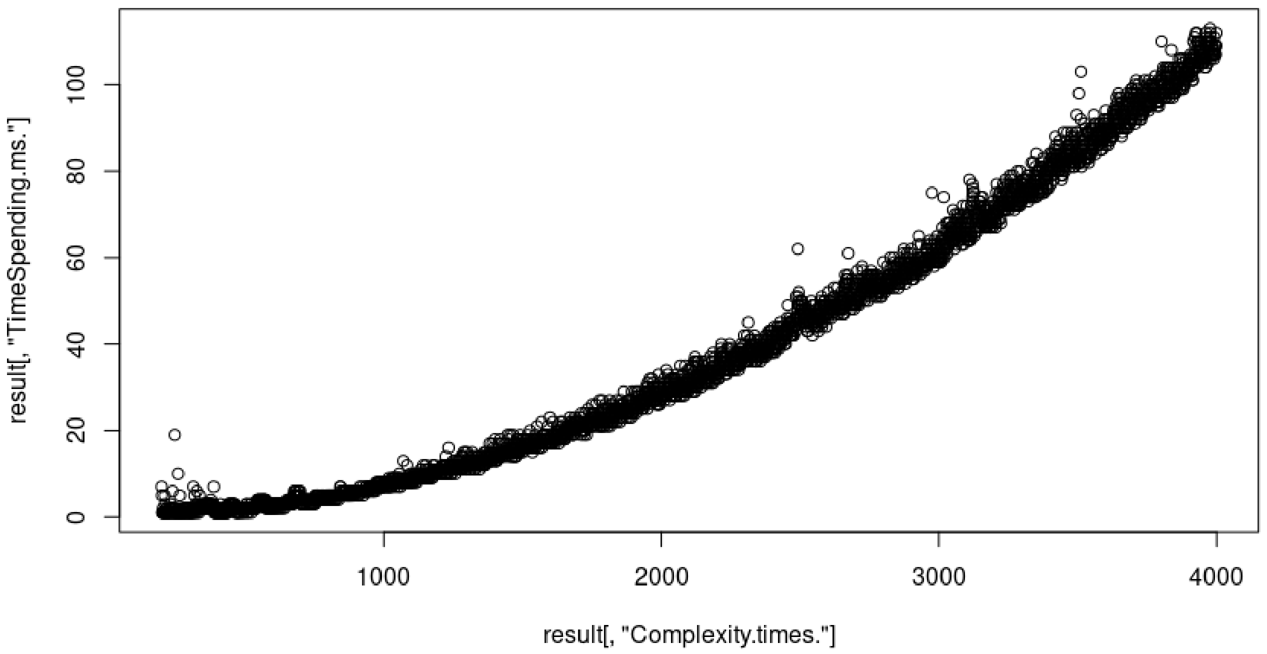
To prove that the result is not caused by the system, I added the size of the list to 5000 and try again:



The result still follows. The third try I tried to add the complexity to 3000 but minus the size of list to 1000, and R export the graph:

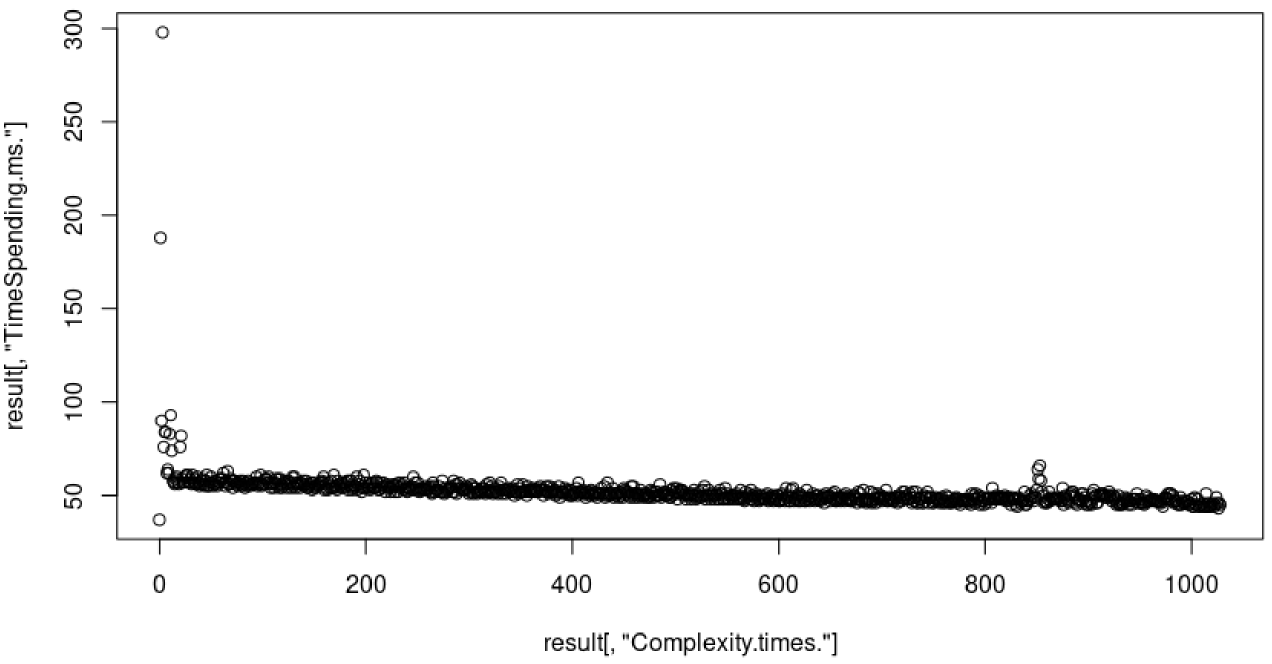


It’s quite confusing but the truth is that as the complexity goes higher, time spending will goes slower. However I found that with the size of the list goes up, the time spending would also goes up, So I have made the experiment:
I picked up the size of the list from 0 to 3000 and for each of them I randomly sorted them 100 times, and pick up the time spending:



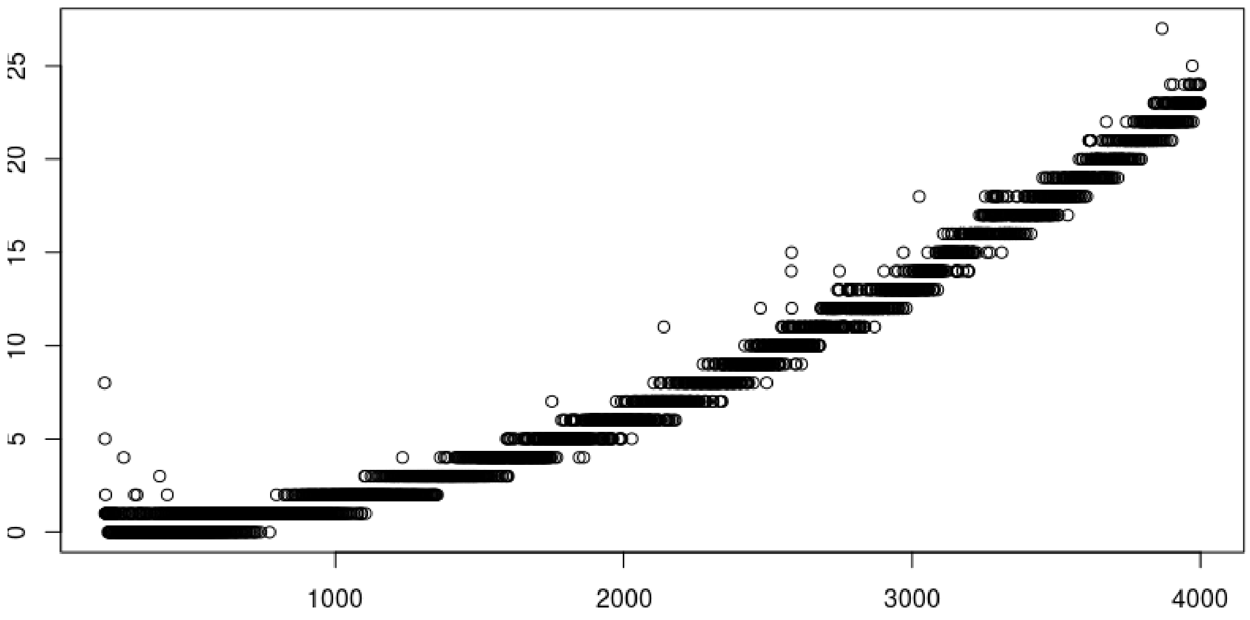
It is obvious that as the size of the list goes higher and the higher the size is, the speed if increasing time spending will goes higher, the time spending for sorting the list also goes higher such that we can draw a conclusion that time spending for quick sort will goes less as the complexity goes up, but the time spending will goes more when the size of the list goes up.

2.2 Bubble sort:

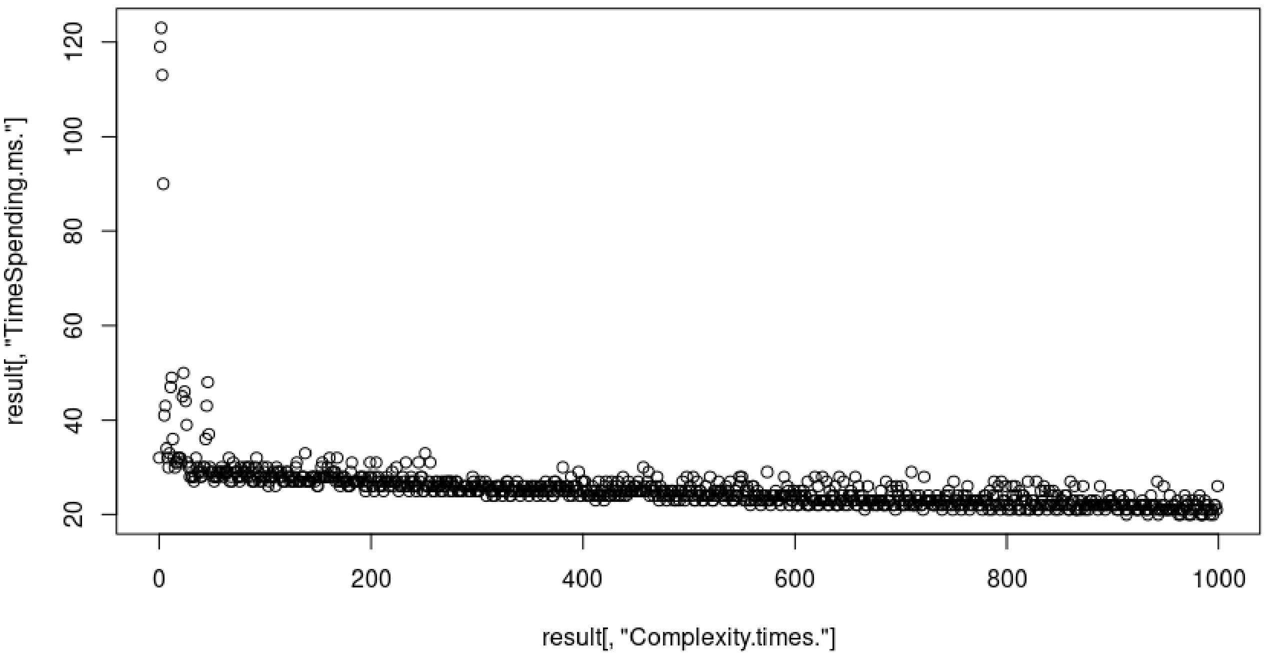


Graph above is the graph of bubble sort which sorting the size of the list is 3000 and the max complexity is 1000. The graph is similar to quick sort that time spending for quick sort will goes

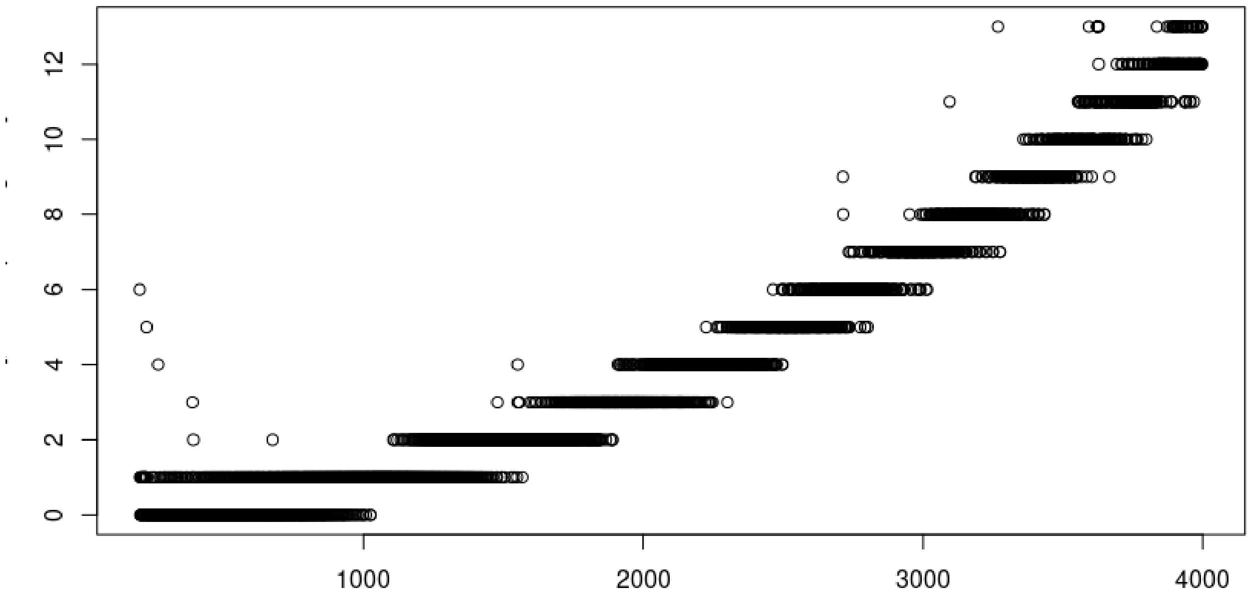
less as the complexity goes up. By comparing bubble sort with quick sort with same complexity and same size of list, we can find generally quick sort is slightly faster than bubble sort.
Time spending for bubble sort is positive related to the size of the list and the shape of the bubble sort is like a stair.



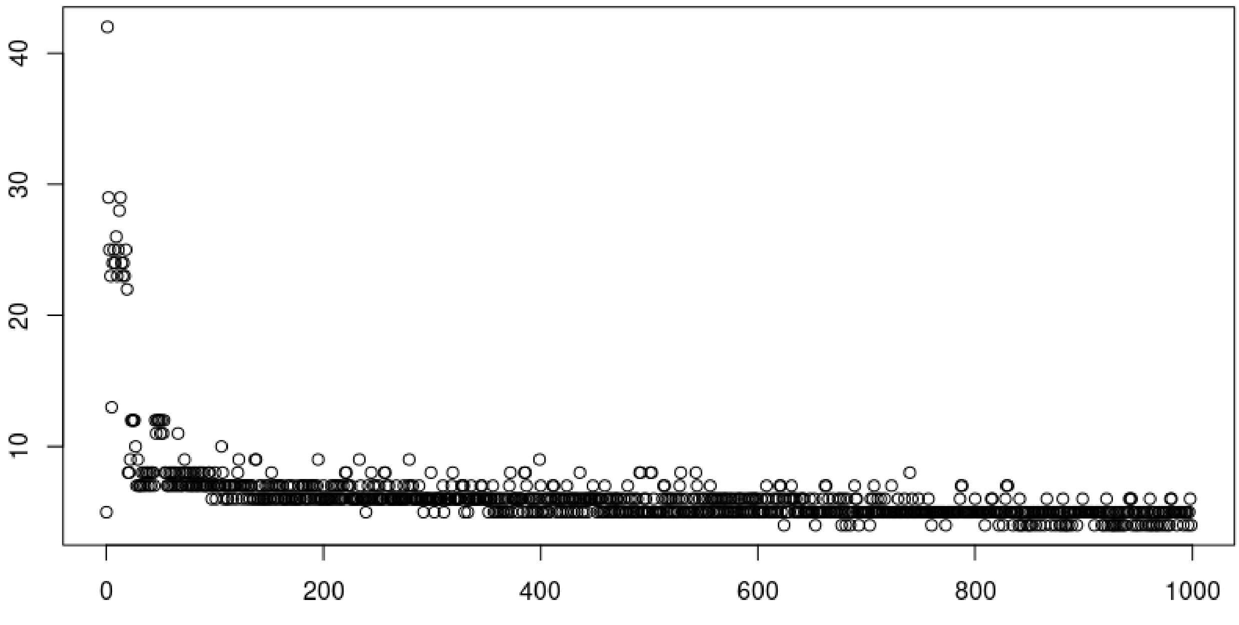
2.3 Selection Sort:
Setup of the selection sort is the same as the bubble sort and the first grapgh of the quick sort. Similar to both bubble sort and quick sort, time spending for selection sort tends to decreasing as complexity goes higher. However we can also find selection sort seems faster than both bubble sort and quick sort.



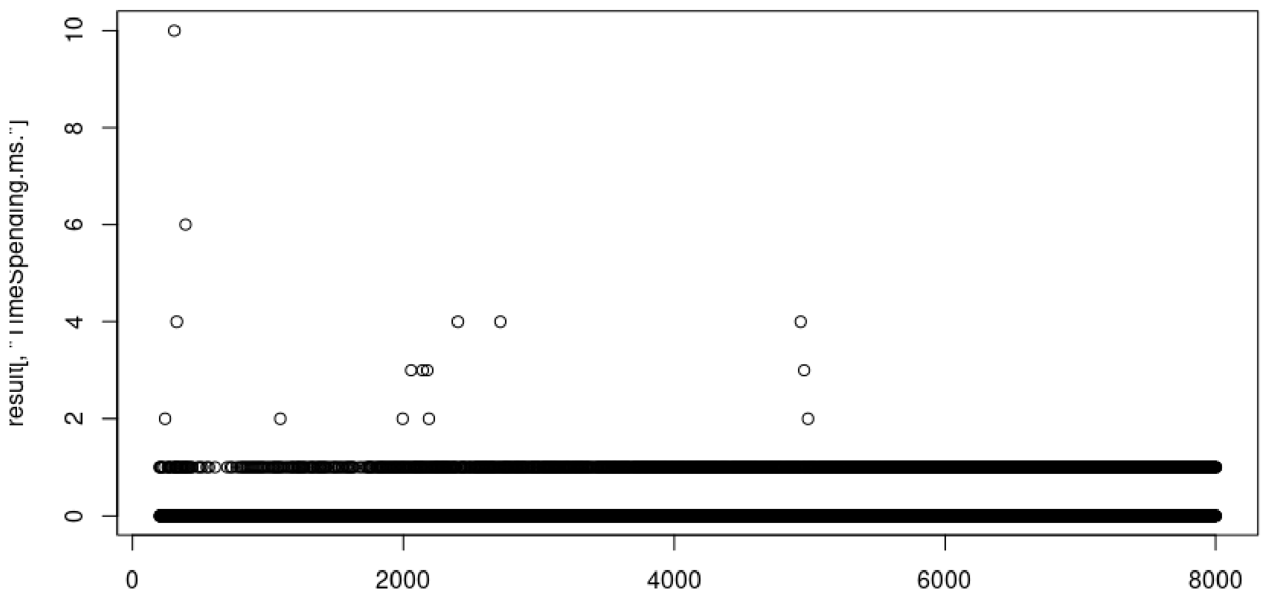
And the size is also positive related with time spending but the average time is far less than the time spending on bubble sort.



2.4 Merge Sort:
As all of the element inside of the merge sort have to be inserted to temp list, time spending of merge sort tends to be changeless.



However, as the merge sort is operated using index only, the relationship between size of the list and time spending seems meaningless:



As the time complexity are always in a very short time because the operation happens to index instead of the list.

Conclusion:
By analysing these of three kinds of sorting method, I found that all of data I have analysed are not inaccuracy that even if I am trying to prevent the systematic errors to get a more accurate data, however the program needs to read through all of the list and choose or swap elements in order to sort them. However we can find that time spending on sorting the program is always keeping in a range and the time spending on sorting the program(the most obvious one is 3rd graph of the quick sort) will index increasing as the size increasing.