DATA-Lab7

Question 2.

The time complexity of each function is as followed:

1. quickSort(): O(nlog(n))
2. mergeSort(): O(nlog(n))
3. heapSort(): O(nlog(n))
4. insertionSort(): O(n^2)

⇒ ***Expectation:*** T_insertion > T_quick ~ T_merge ~ T_heap

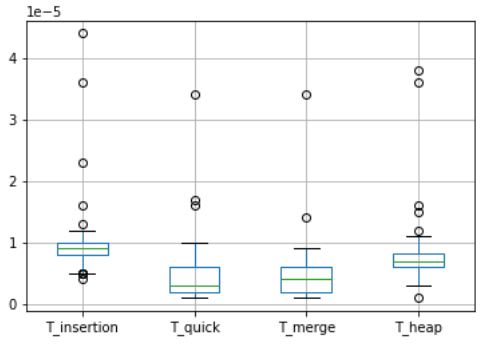
Method:

1. The running time of each function on an array of 1000 integers is collected and plotted.
2. Wilcoxon one sided or two sided tests are performed suitable with the result in the diagram.

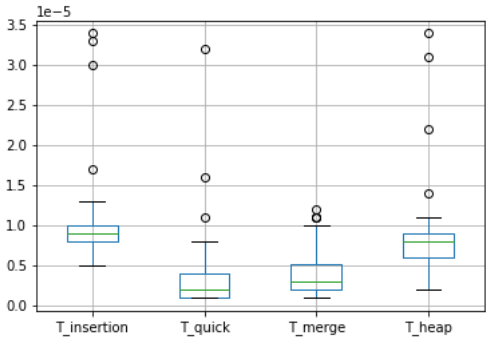
As you can see from 5 attempts below, insertionSort() is the slowest algorithm, followed by heapSort(). However, it is hard to tell whether quickSort() or mergeSort() is faster. I performed a two-sided Wilcoxon test with ***H0*** that there is no significant difference in running time of quick Sort() and mergeSort(). In 3 out of 5 attempts, the H0 was not rejected (pvalue > 0.05). Therefore, I performed 500 more Wilcoxon tests and take the major vote as the result. How I count the cases is presented in the flowchart below.

⇒ The result shows that around 9.0% of the tests, quickSort() is faster than mergeSort() and 0.4% in the reverse way. This implies that around 90.6% of the tests, there is no significant difference in performance of quickSort() and mergeSort().

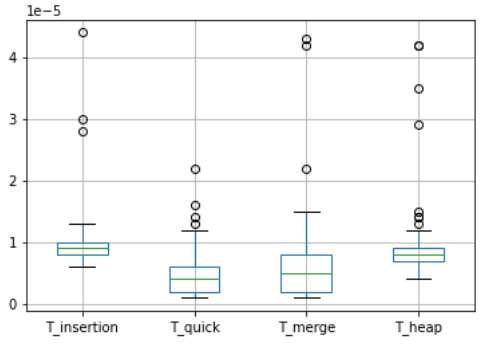
To conclude, for an array of 1000 numbers, either quickSort() or mergeSort() is the best algorithm for use.



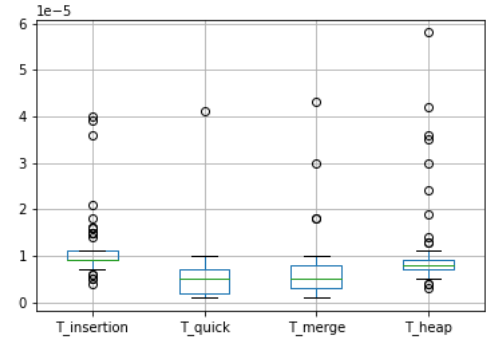
Attempt no.1



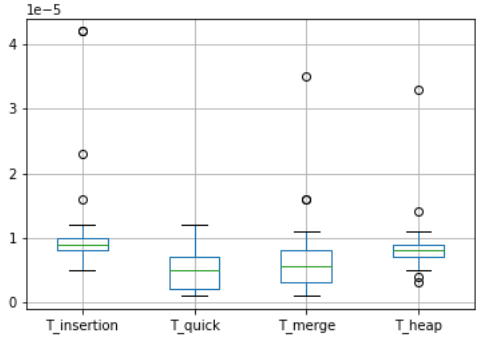
Attempt no.2



Attempt no.3

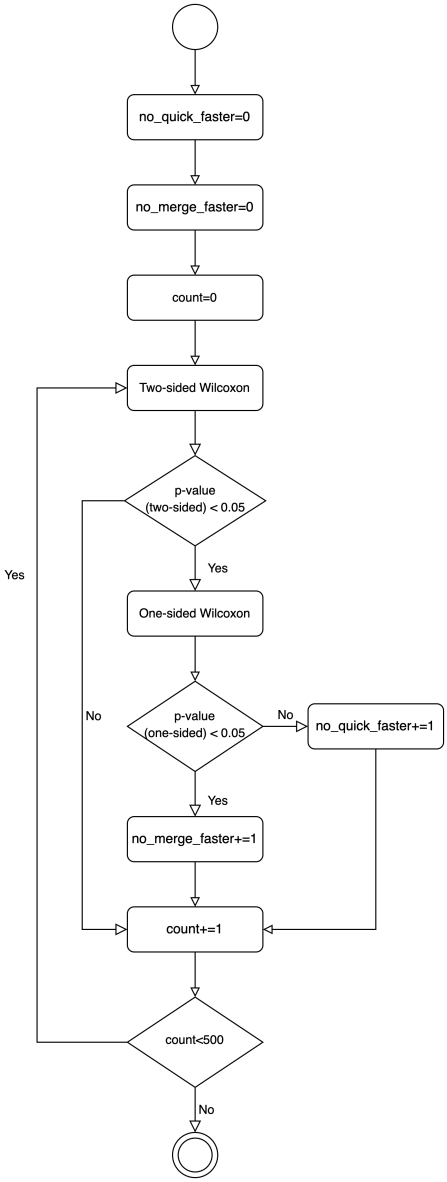


Attempt no.4



Attempt no.5

```
Wilcoxon two-sided
Attempt no.1
   T_quick-T_merge pvalue: 0.2458
Attempt no.2
   T_quick-T_merge pvalue: 0.0107
Attempt no.3
   T_quick-T_merge pvalue: 0.0286
Attempt no.4
   T_quick-T_merge pvalue: 0.0571
Attempt no.5
   T_quick-T_merge pvalue: 0.0608
```




```
%quickSort() faster: 9.00
%mergeSort() faster: 0.40
```

Question 3

In order to make a good combination of sorting algorithms, I find the optimal range for each algorithm, ie. I find the key point that a functions starts consistently running faster than another.

Although the heapSort() runs in O(nlogn), not until around 440 that it consistently starts continuously running faster than insertionSort() which works in O(n^2). Moreover, it is slower than quickSort() and merge() in previous analysis ⇒ I decided not to use heapSort in the combination.

Result

- n ≤ 30: T_insertion < T_merge < T_quick
- n=[30,50]: T_insertion < T_quick < T_merge
- n=[50,90]: T_quick < T_insertion < T_merge
- n ≥ 90: T_quick < T_merge < T_insertion

⇒ Combination_sort():

- n < 50: insertionSort()
- n ≥ 50: quickSort()

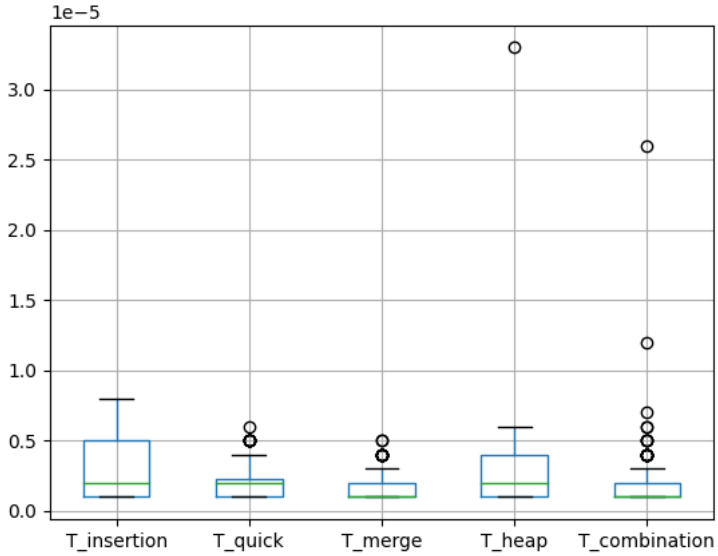
To compare with individual algorithms, I performed Wilcoxon one-sided tests with H0 that T_ combination is significantly smaller than each of other sorting algorithms.

For 5 attempts, all p_value are greater than 0.05, so H0 is not rejected.

```
Result for the comparision of heapSort() and insertionSort()
maxSize: 1000
Attempt no1.
        [492, 532, 456, 487, 451, 461, 472, 481, 465, 462, 440, 442, 480, 455, 479, 430, 439, 435, 425, 409, 435, 418, 463, 435, 440,
         Approximately n = 443, <function heapSort at 0x119b21ee0> is the faster than <function insertionSort at 0x119b21b80>
Attempt no2.
        [646, 432, 448, 455, 475, 468, 404, 485, 423, 411, 449, 486, 418, 460, 426, 441, 445, 414, 463, 500, 475, 421, 424, 422, 415,
        Approximately n = 442, <function heapSort at 0x114ca5ee0> is the faster than <function insertionSort at 0x114ca5b80>
```

```
Result for the comparision of insertionSort(), mergSort() and quickSort()
maxsize: 100
Attempt no1.
        [77, 70, 83, 71, 108, 85, 88, 87, 80, 136, 74, 83, 92, 133, 83, 121, 82, 105, 87, 75, 75, 66, 77, 71, 66, 77, 79, 78, 72, 70,
        Approximately n = 81, <function mergeSort at 0x11d510c10> is the faster than <function insertionSort at 0x11d510af0>
        [54, 54, 29, 9, 4, 2, 1, 153, 6, 9, 1, 2, 3, 103, 2, 2, 6, 54, 17, 2, 4, 5, 1, 5, 58, 90, 5, 5, 104, 30, 3, 97, 2, 137, 13, 65
        Approximately n = 38, <function quickSort at 0x11d510700> is the faster than <function mergeSort at 0x11d510c10>
        [45, 29, 42, 32, 47, 38, 82, 62, 60, 55, 43, 47, 60, 44, 48, 38, 52, 75, 65, 66, 97, 47, 66, 94, 41, 36, 41, 79, 60, 60, 53, 4
        Approximately n = 52, <function quickSort at 0x11d510700> is the faster than <function insertionSort at 0x11d510af0>
Attempt no2.
        [92, 86, 92, 96, 80, 75, 84, 80, 64, 74, 67, 75, 77, 73, 70, 89, 96, 90, 86, 96, 76, 75, 69, 81, 74, 90, 72, 76, 71, 84, 75, 6
        Approximately n = 80, <function mergeSort at 0x119a7ac10> is the faster than <function insertionSort at 0x119a7aaf0>
        [40, 8, 1, 8, 23, 2, 8, 5, 48, 3, 4, 7, 89, 50, 97, 45, 97, 56, 6, 76, 4, 96, 94, 12, 1, 83, 84, 41, 5, 3, 82, 98, 4, 4, 52, 1
        Approximately n = 37, <function quickSort at 0x119a7a700> is the faster than <function mergeSort at 0x119a7ac10>
        [74, 40, 32, 38, 31, 32, 37, 35, 76, 40, 63, 41, 36, 31, 37, 32, 53, 44, 37, 55, 40, 33, 33, 41, 39, 43, 40, 35, 95, 82, 32, 3
        Approximately n = 42, <function quickSort at 0x119a7a700> is the faster than <function insertionSort at 0x119a7aaf0>
```

```
Wilcoxon one-sided test between combination_sort and other sorting algorimths
Attempt no.1
Done generating data
        n: 50_100_500_1000_10000
                T_insertion pvalue: 1.0000
                T_quick pvalue: 0.6515
                T_merge pvalue: 0.1044
                T_heap pvalue: 0.9828
Attempt no.2
Done generating data
        n: 50_100_500_1000_10000
                T_insertion pvalue: 1.0000
                T_quick pvalue: 0.3361
                T_merge pvalue: 0.3182
                T_heap pvalue: 0.9998
Attempt no.3
Done generating data
        n: 50_100_500_1000_10000
                T_insertion pvalue: 1.0000
                T_quick pvalue: 0.7835
                T_merge pvalue: 0.4411
                T_heap pvalue: 1.0000
Attempt no.4
Done generating data
        n: 50_100_500_1000_10000
                T_insertion pvalue: 1.0000
                T_quick pvalue: 0.6372
                T_merge pvalue: 0.3622
                T_heap pvalue: 0.9962
Attempt no.5
Done generating data
        n: 50_100_500_1000_10000
                T_insertion pvalue: 1.0000
                T_quick pvalue: 0.0865
                T_merge pvalue: 0.6517
                T_heap pvalue: 0.9794
```


1e−5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
T_insertion
T_quick
T_merge
T_heap
T_combination