Problem #1

Problem 1

Order from lowest to highest rate.

37

$\sqrt{N}$

$N/2$, N → grow at the same rate.

$$\lim_{N\to\infty} \frac{N/2}{N} = \frac{1}{2} > 0$$

$N \log\log N$

$N \log N$, $N \log(N^2)$ → grow at the same rate.

$$\lim_{N\to\infty} \frac{N\log N}{N\log(N^2)} = \lim_{N\to\infty} \frac{N\log N}{2N\log N} = \frac{1}{2} > 0$$

$N \log^2 N$ $(= N(\log N)(\log N))$

$N^{1.5}$ $(= N\sqrt{N})$

N^2

$N^2 \log N$

N^3

$2^{\frac{N}{2}}$ $(= \sqrt{2^N})$

2^N

Problem #2

Problem 2.

Let $n \log n > n^{1+\frac{\varepsilon}{\sqrt{\log n}}}$, $\varepsilon < 0$.

$$n \log n > n \cdot n^{\frac{\varepsilon}{\sqrt{\log n}}}$$

$$\log n > n^{\frac{\varepsilon}{\sqrt{\log n}}}$$

$$\log(\log n) > \log n^{\frac{\varepsilon}{\sqrt{\log n}}}$$

$$\log(\log n) > \frac{\varepsilon}{\sqrt{\log n}} \log n = \varepsilon \cdot (\log n)^{-\frac{1}{2}} \cdot \log n$$

$$\log(\log n) > \varepsilon \sqrt{\log n}$$

Let $\log n = A$

$$\log A > \varepsilon \sqrt{A}$$

This means $\log A$ grows faster than $\varepsilon \sqrt{A}$.

$\lim_{A \to \infty} \frac{\log A}{\varepsilon \sqrt{A}} = 0$, so $\log A = O(\varepsilon \sqrt{A})$

$\log(\log n) = O(\varepsilon \sqrt{\log n})$

This is a contradiction to the assumption that $n \log n > n^{1+\frac{\varepsilon}{\sqrt{\log n}}}$

$\therefore n^{1+\varepsilon/\sqrt{\log n}}$ grows faster than $n \log n$.

Problem #3

Problem #3

(1)
```
sum = 0;                          → constant time C1 for some constant C1
for (i=0; i<n; ++i)               → linear time C2n for some constant C2
    ++sum;
```

total running time: $C_1 + C_2 n$

big-O time: $O(n)$

(2)
```
sum = 0;                          → C1
for (i=0; i<n; ++i)
    for (j=0; j<n; ++j)           → C2n^2 for some constant C1, C2
        ++sum;
```

total running time: $C_1 + C_2 n^2$

big-O time: $O(n^2)$

(3)
```
sum = 0;                          → C1
for (i=0; i<n!; ++i)
    for (j=0; j<n*n; ++j)         → C2·(n)·(n^2) for some constant C1, C2
        ++sum
```

total running time: $C_1 + C_2 n^3$

big-O time: $O(n^3)$

(4) sum = 0; → C_1

```
for (i=0; i<n; ++i)
  for (j=0; j<i; ++j)
    ++sum;
```

i=0	0
i=1, j=0	1
i=2, j=0, 1	2
i=3, j=0, 1, 2	3
i=4, j=0, 1, 2, 3	4
⋮	⋮
i=n j=0, 1, 2, 3, ..., n-2, n-1	n-1

$\Rightarrow 0+1+2+3+\cdots+(n-2)+(n-1) = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$

so $C_2\left(\frac{n^2-n}{2}\right)$

where C_1 C_2 are const

Total running time : $C_1 + C_2\left(\frac{n^2-n}{2}\right)$

big-O time : $O(n^2)$

(5)
```
sum=0;                                   → C1
for(i=0; i<n; ++i)
  for(j=0; j<i*i; ++j)
    for(k=0; k<j; ++k)
      ++sum;
```

i=0 → 0

i=1 j=0 → 0

i=2 j=0, 1, 2, 3 → 1+2+3 = 6
k= 0 | 0,1 | 0,1,2

i=3 j=0, 1, 2, 3, 4, 5, ···, 7, 8
k= 0 | 0,1 | 0,1,2 | 0~6 | 0~7
→ 1+2+3+ ··· +7+8 = $\frac{8\cdot 9}{2} = 36$

i=4 j=0, 1, 2, 3, ···, 14, 15
k= 0 | 0,1 | 0~13 | 0~14
→ 1+2+3+ ··· 14+15 = $\frac{15\cdot 16}{2} = 120$

⋮

i=n j=0, 1, 2 ··· n^2-2, n^2-1
k= 0 | 0,1 | $0\sim n^2-3$ | $0\sim n^2-2$
→ $1+2+3+\cdots+n^2-2+n^2-1 = \frac{(n^2-1)\,n^2}{2} = \frac{n^4-n^2}{2}$

$$\Rightarrow 0+0+\frac{3\cdot 4}{2}+\frac{8\cdot 9}{2}+\frac{15\cdot 16}{2}+\cdots+\frac{(n^2-1)n^2}{2}$$

$$= \frac{1}{2}\sum_{i=2}^{n}(i^2-1)i^2 = \frac{1}{2}\left[\frac{1}{10}(n-1)n(n+1)(n+2)(2n+1)\right]$$

※ Summation computed by WolframAlpha

Total running time : $C_1 + C_2\left[\frac{1}{20}(n-1)n(n+1)(n+2)(2n+1)\right]$

big-O time : $O(n^5)$

(6)
```
sum=0;                                   → C1
for(i=1; i<n; ++i)
    for(j=1; j<i*i; ++j)
        if(j%i==0)
            for(k=0; k<j; ++k)
                ++sum;
```

i=1

i=2, j=1, 2, 3, → 2
k= 0, 1

i=3, j=1, 2, 3, 4, 5, 6, 7, 8 → 3 + 6 = 9
k= 0,1,2 0~5
3·1 3·2

i=4, j=1, 2, 3, 4, 5, ··· 8, ··· 12 ···, 15 → 4 + 8 + 12 = 24
k= 0~3 0~7 0~11
4·1 4·2 4·3

i=5, j=1, ··· 5 ··· 10 ··· 15 ··· 20 → 5 + 10 + 15 + 20 = 50
k= 0~4 0~9 0~14 0~19
5·1 5·2 5·3 5·4

⋮

i=n, j=1 ··· n ··· 2n ··· 3n ··· (n-1)n
k= 0~n-1 0~2n-1 0~3n-1 0~(n-1)n-1

→ n + 2n + 3n + ··· + (n-1)n = n(1+2+3+···+n-1)

$$= n\frac{(n-1)n}{2} = \frac{n^2(n-1)}{2}$$

$$\Rightarrow (2) + (3+6) + (4+8+12) + \cdots\cdots + \frac{n^2(n-1)}{2}$$

$$= \frac{1}{2}\sum_{i=2}^{n} i^2(i-1) = \frac{1}{24}(n-1)n(n+1)(3n+2)$$

※ summation computed by WolframAlpha

Total running time : $C_1 + C_2\left[\frac{1}{24}(n-1)(n)(n+1)(3n+2)\right]$

big-O time : $O(n^4)$

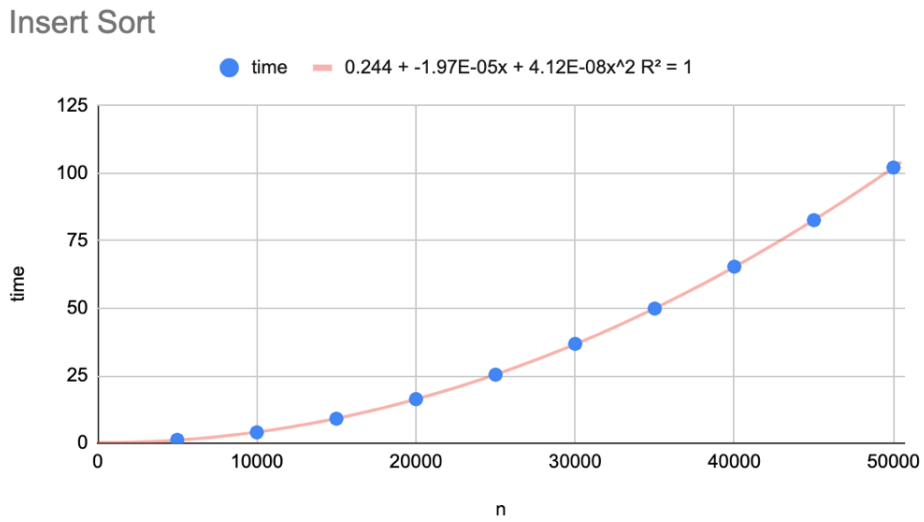
Problem #5.b

Each time is an average time of 3 times runs for each value

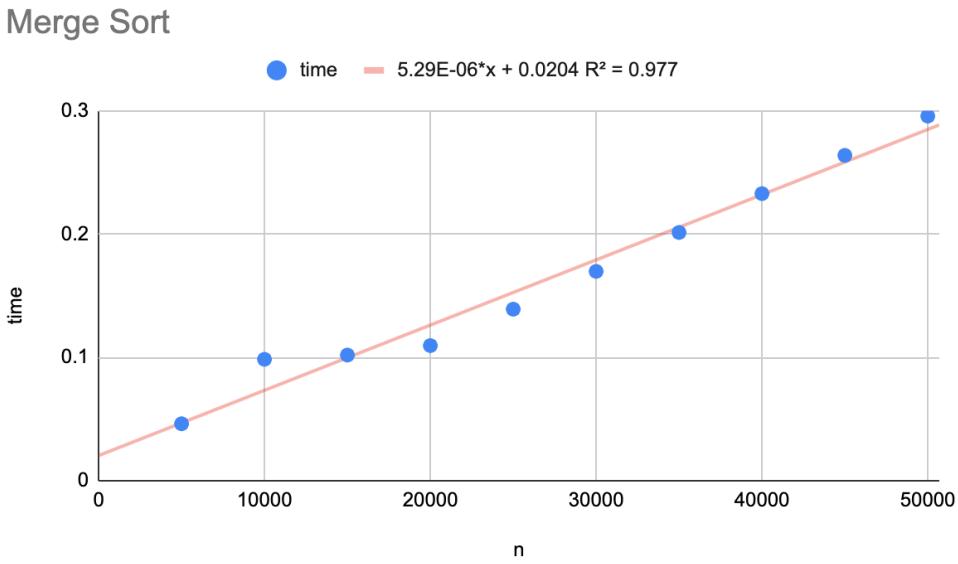
Insertion Sort	
n	time
5000	1.28704722722
10000	4.06039706866
15000	9.11580936114
20000	16.3499266307
25000	25.4163123767
30000	36.8228400548
35000	49.8924613794
40000	65.4234353701
45000	82.6217943033
50000	102.122803688

Merge Sort	
n	time
5000	0.0461699962616
10000	0.0984280109406
15000	0.101870934168
20000	0.109541257222
25000	0.139062245687
30000	0.169729312261
35000	0.201289653778
40000	0.232779105504
45000	0.263806025187
50000	0.295650959015

Problem #5.c



f(x) = 4.12E-08x^2 + -1.97E-05x + 0.244, a polynomial of degree 2, best fits to this graph where x is size of array, f(x) is time



f(x) = 5.29E-06x + 0.0204, a linear curve, fits to this graph
where x is size of array, f(x) is time. However, f(x) = axLog(x) where a > 0 would best fit to the data when there are large quantities of data as input.

Problem #5.d

Problem #5.e

We need to apply average cases for both sorting algorithms since we used data of randomly generated integers which are statistically distributed in each list.

For Insertion sort, experimental running time was calculated to 4.12E-08x^2 + -1.97E-05x + 0.244. By dropping lower terms and constants, we are given running time O(x^2). Theoretical running time of insertion sort for average-case is O(x^2). Thus, there's no difference between two running times.

For Merge sort, following equation was computed by data f(x) = 5.29E-06x + 0.0204 and the running time is O(x) which is different than O(xLogx), theoretical running time of merge sort for average-case. However, as mentioned at #5c, axLogx best fits to the merge sort program eventually, and the running time complexity is O(xLogx) which is the same with the theoretical running time. Therefore, there's no difference between two running times.

<Extra Credit>

Insertion sort

Best-case: Sorting sorted array, [1, 2, 3, …., n-1, n], O(n)
Worst case: Sorting reversely sorted array, [n, n-1, n-2,…., 2, 1], O(n^2)

I created best-case lists with size of n by placing integers from 1 to n into each list in order, and created worst-case lists by simply reversing the sorted list.

Merge sort

Theoretically, running times of merge sort for all cases(best, worst, and average) are the same and there are only insignificant difference in constants. But the best and worst case of merge sort do exist like below.

Best-case: Sorting sorted array, [1, 2, 3, …., n-1, n], O(nlogn),
the minimum number of comparisons.
Worst-case: Sorting array in certain order that would require the maximum number of comparisons.

Let's say,
a is real number
N is a number label for a number

There's a series of numbers,
a0 <= a1 <= a2 <= a3 <= a4 <= a5 <= a6 <= a7 …….. a(N-1) <= a(N)

[
a4, a0, a6, a2, a12, a8, a14, a10, ….. ,a(N-3), a(N-7), a(N-1), a(N-5),
a5, a1, a7, a3, a13, a9, a15, a11, ….. , a(N-2), a(N-6), a(N), a(N-4)
]

Above list would guarantee the maximum number of comparisons when using a merge sort.

However, because such lists with this pattern is not common in the real world, for this assignment I've decided to use the same examples with ones used for insertion sort, sorted list and reversely sorted list, to verify if there's significant difference in running times when sorting two different lists, and it would be more beneficial to compare two sorting algorithms with the identical cases.

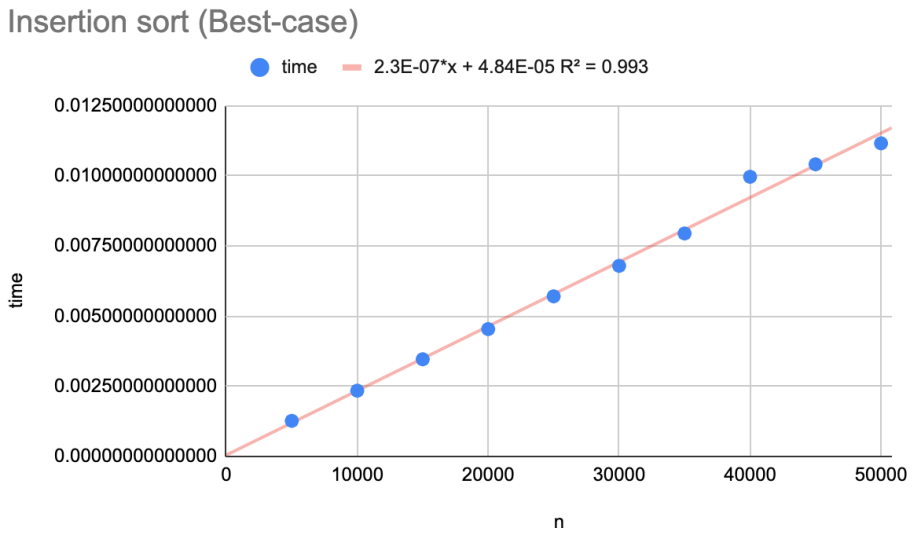
Best-case: Sorting reversely sorted array, [n, n-1, n-2,…., 2, 1], O(nlogn)
Worst-case: Sorting sorted array, [1, 2, 3, …., n-1, n], O(nlogn)

E.C. b)

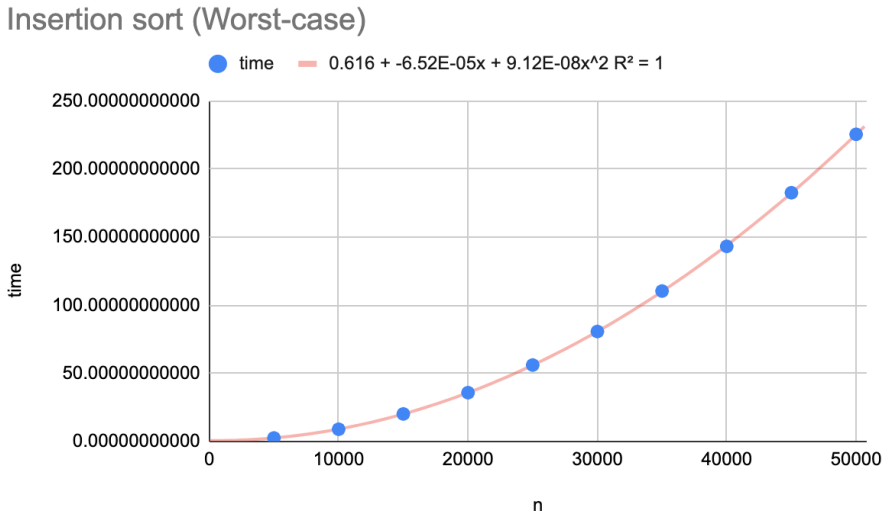
Insertion Sort Best-case		Insertion Sort Worst-case	
n	time	n	time
5000	0.00126814842224	5000	2.57639312744
10000	0.00234293937683	10000	8.98389196396
15000	0.00346302986145	15000	20.2190179825
20000	0.00453901290894	20000	35.8310611248
25000	0.00570607185364	25000	56.1140508652
30000	0.00679302215576	30000	80.7515780926
35000	0.00794696807861	35000	110.454810858
40000	0.00996708869934	40000	143.267274141
45000	0.0104119777679	45000	182.650673151
50000	0.0111601352692	50000	225.541687965

Merge Sort Best-case		Merge Sort Worst-case	
n	time	n	time
5000	0.0203080177307	5000	0.0214800834656
10000	0.043200969696	10000	0.0444052219391
15000	0.0664930343628	15000	0.0683648586273
20000	0.0917270183563	20000	0.0936851501465
25000	0.116053819656	25000	0.118448972702
30000	0.140550136566	30000	0.143771886826
35000	0.167263984680	35000	0.170320987701
40000	0.193044900894	40000	0.197106122971
45000	0.219304084778	45000	0.222460031509
50000	0.244823932648	50000	0.248453855515

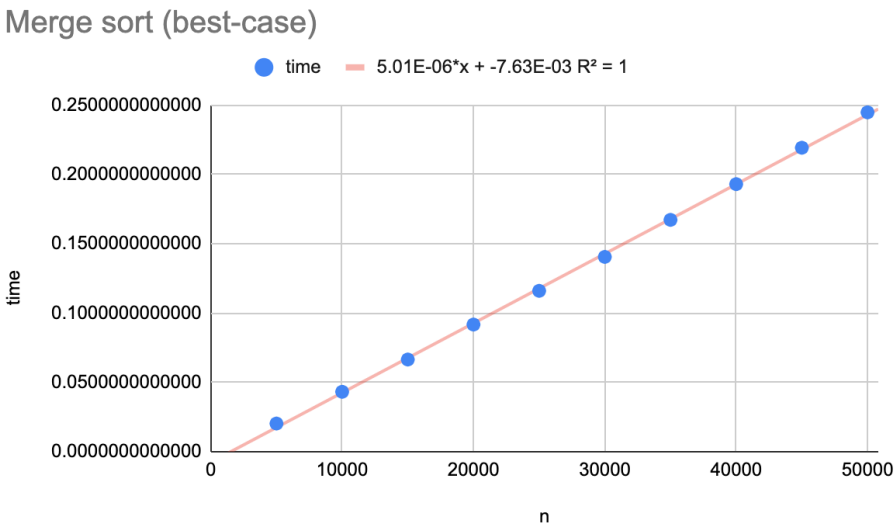
E.C. c)



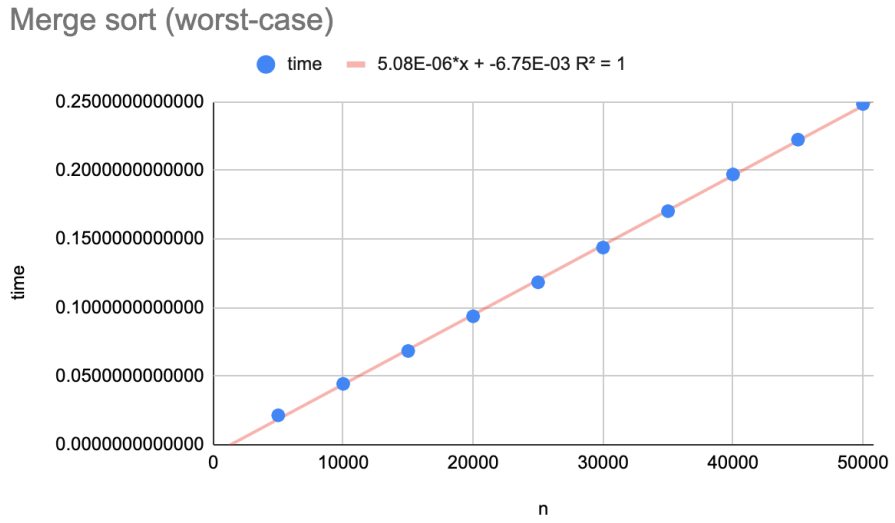
f(x) = 2.3E-07x + 4.84E-05, a linear curve, best fits to this graph



f(x) = 9.12E-08x^2 + -6.52E-05x + 0.616, a polynomial of degree 2, best fits to this graph

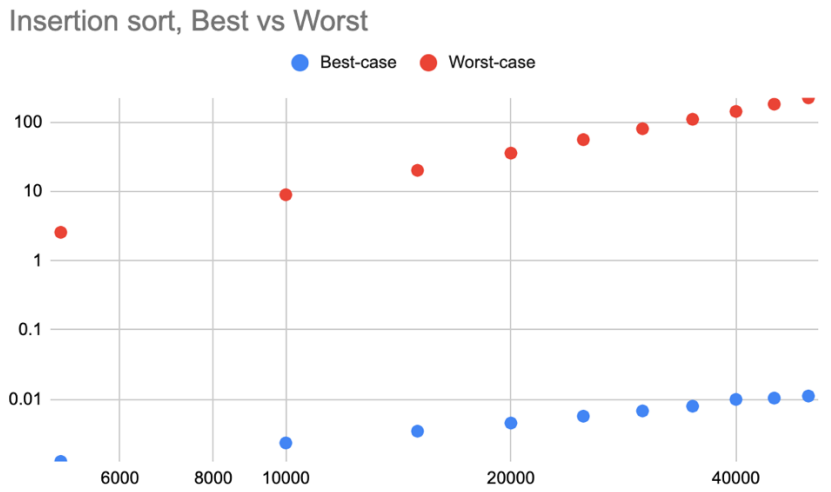


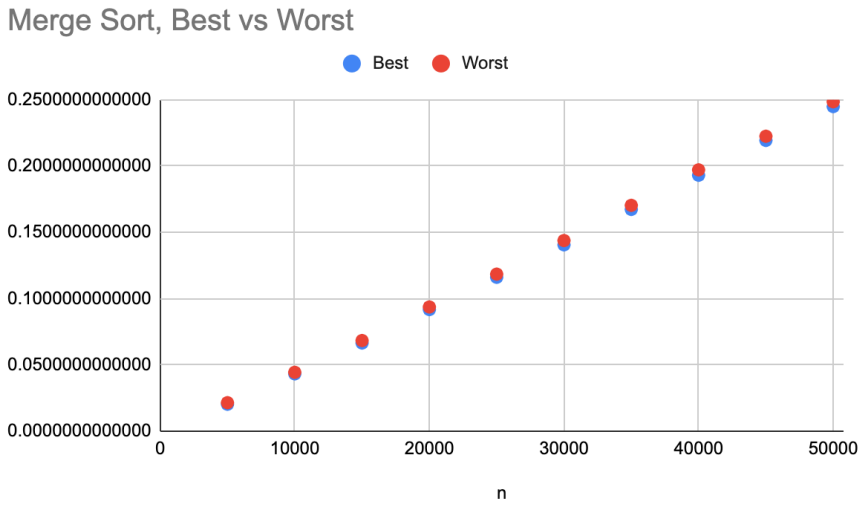
f(x) = 5.01E-06x + -7.63E-03, a linear curve, best fits to this graph. However, f(x) = axLog(x) where a > 0 would best fit to the data when there are large quantities of data as input.



f(x) = 5.08E-06x + -6.75E-03, a linear curve, best fits to this graph. However, f(x) = axLog(x) where a > 0 would best fit to the data when there are large quantities of data as input.

E.C. d)


Insertion sort, Best vs Worst
Best-case
Worst-case
100
10
1
0.1
0.01
6000
8000
10000
20000
40000


Merge Sort, Best vs Worst
Best
Worst
0.2500000000000
0.2000000000000
0.1500000000000
0.1000000000000
0.0500000000000
0.0000000000000
0
10000
20000
30000
40000
50000
n

E.C. e)

Insertion sort
Best-case: Theoretically, best-case of this algorithm takes O(n). Experiment calculated a linear curve and its complexity is O(n). Therefore, there's no difference between theory and experiment.
Worst-case: Theoretically, worst-case of this algorithm takes O(n^2). Experiment calculated a polynomial of degree 2 and its complexity is O(n^2). Therefore, there's no difference between theory and experiment.

Merge sort
Theoretically, running time complexity of both best-case and worst-case of merge sort algorithm is O(nlogn) which is the same with running time O(nlogn) we calculated from the experimental results.

<citations>

Merge Sort Algorithm: https://www.geeksforgeeks.org/merge-sort/
Merge Sort Worst-case: https://stackoverflow.com/questions/24594112/when-will-the-worst-case-of-merge-sort-occur