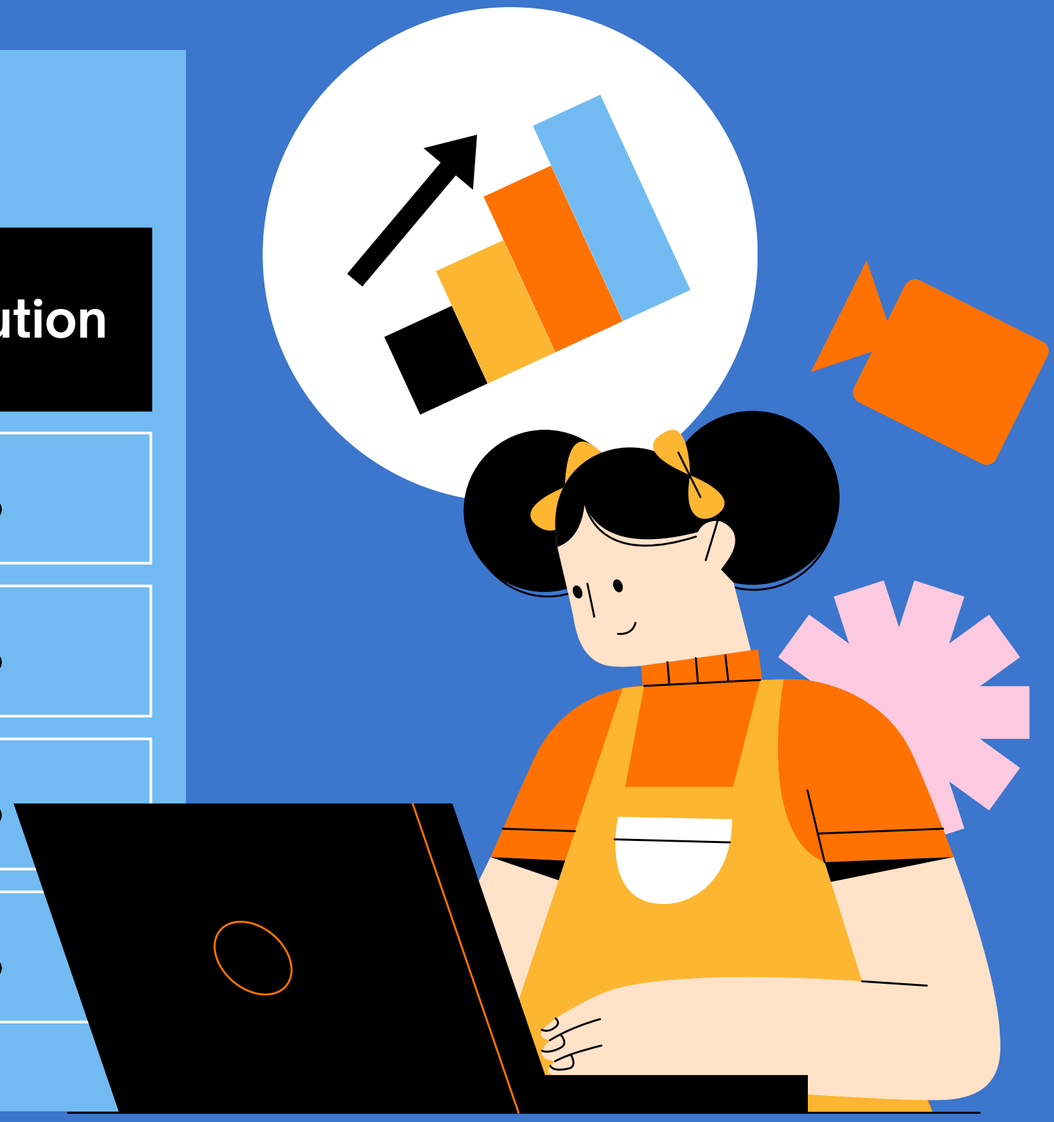
Algorithm Design & Analysis Assignment

LAB SECTION: TT2L (G19)

LAB LECTURER: NEOH KEE LIN

TT2L G9

Student ID	Name	Contribution
1211112069	Tang Wei Xiong	25%
1211108003	Joey Tan Rou Yi	25%
1211107904	Yeong Zi Yan	25%
1211108404	Low Wan Jin	25%



Screenshot of Device Specifications (Windows)

DEVICE SPECS

1211112069 - Tang Wei Xiong

System › About

Storage	Graphics Card	Installed RAM	Processor
1.40 TB	4 GB	24.0 GB	AMD Ryzen 5 5600H with Radeon Graphics
1.15 TB of 1.40 TB used	Multiple GPUs installed	Speed: 3200 MHz	3.30 GHz

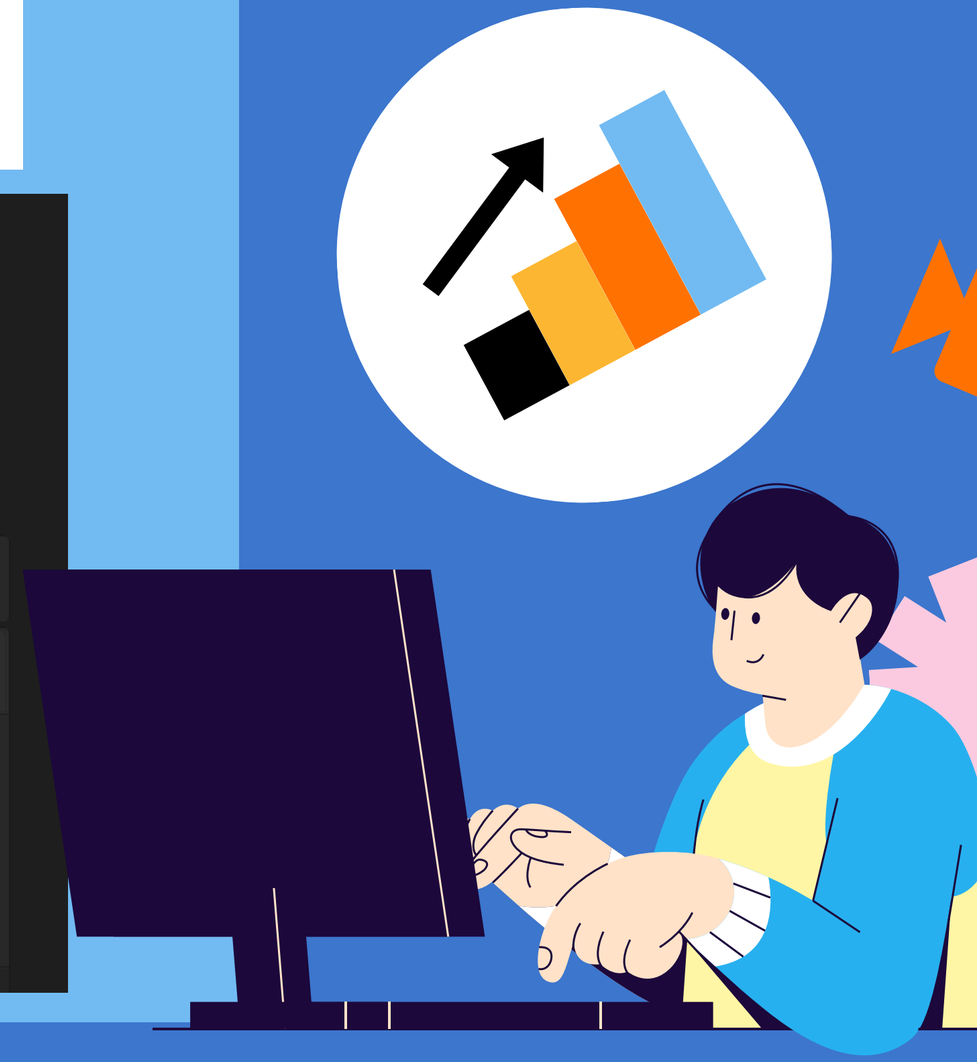
Victus
Victus by HP Laptop 16-e0xxx

Rename this PC

Device specifications

Copy

Device name	Victus	
Processor	AMD Ryzen 5 5600H with Radeon Graphics	3.30 GHz
Installed RAM	24.0 GB (23.3 GB usable)	
Device ID	38A89DD6-6DBF-4D9F-934E-894E039C7862	
Product ID	00356-24539-48831-AAOEM	
System type	64-bit operating system, x64-based processor	
Pen and touch	No pen or touch input is available for this display	



DEVICE SPECS

1211108003 - Joey Tan Rou Yi

存储
477 GB
已使用 477 GB 中的 316 GB

显卡
4 GB
已安装多个 GPU

机带 RAM
24.0 GB
速度: 3200 MHz

处理器
11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz
3.11 GHz

LAPTOP-RPHHABB0
IdeaPad Gaming 3 15IHU6

重命名这台电脑

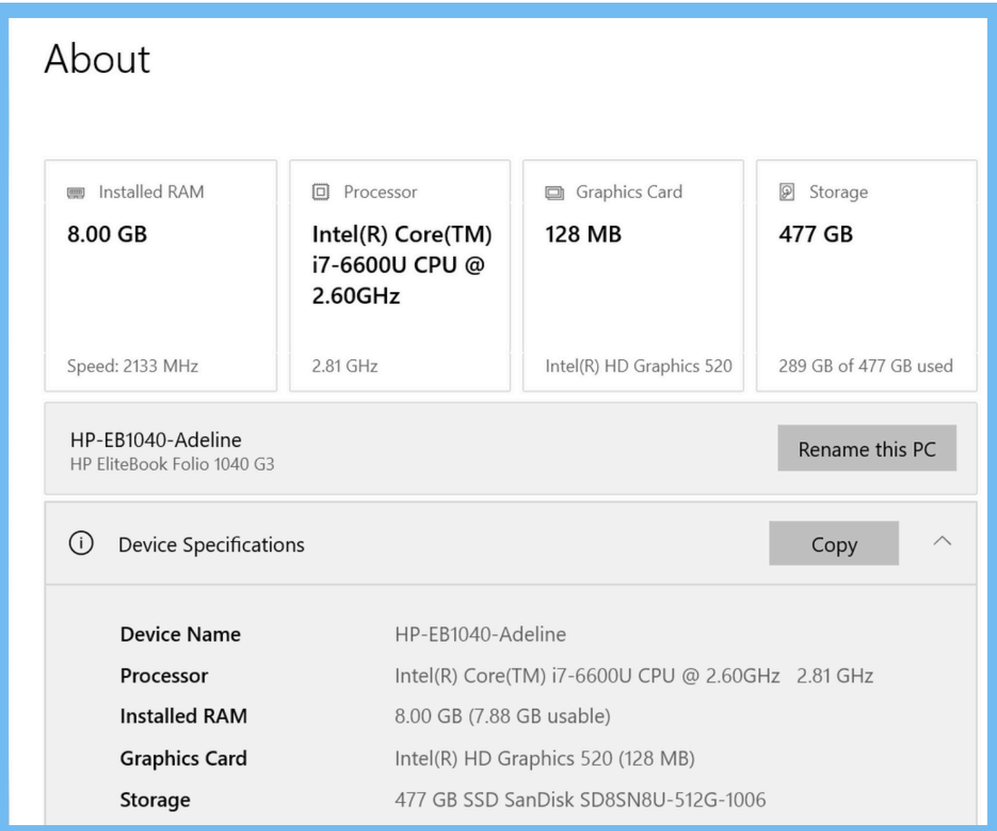
设备规格

复制

设备名称	LAPTOP-RPHHABB0
处理器	11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz 3.11 GHz
机带 RAM	24.0 GB (23.8 GB 可用)
设备 ID	C1AD4162-A72F-4691-899D-347769B10B13
产品 ID	00342-43242-35736-AAOEM
系统类型	64 位操作系统, 基于 x64 的处理器
笔和触控	没有可用于此显示器的笔或触控输入

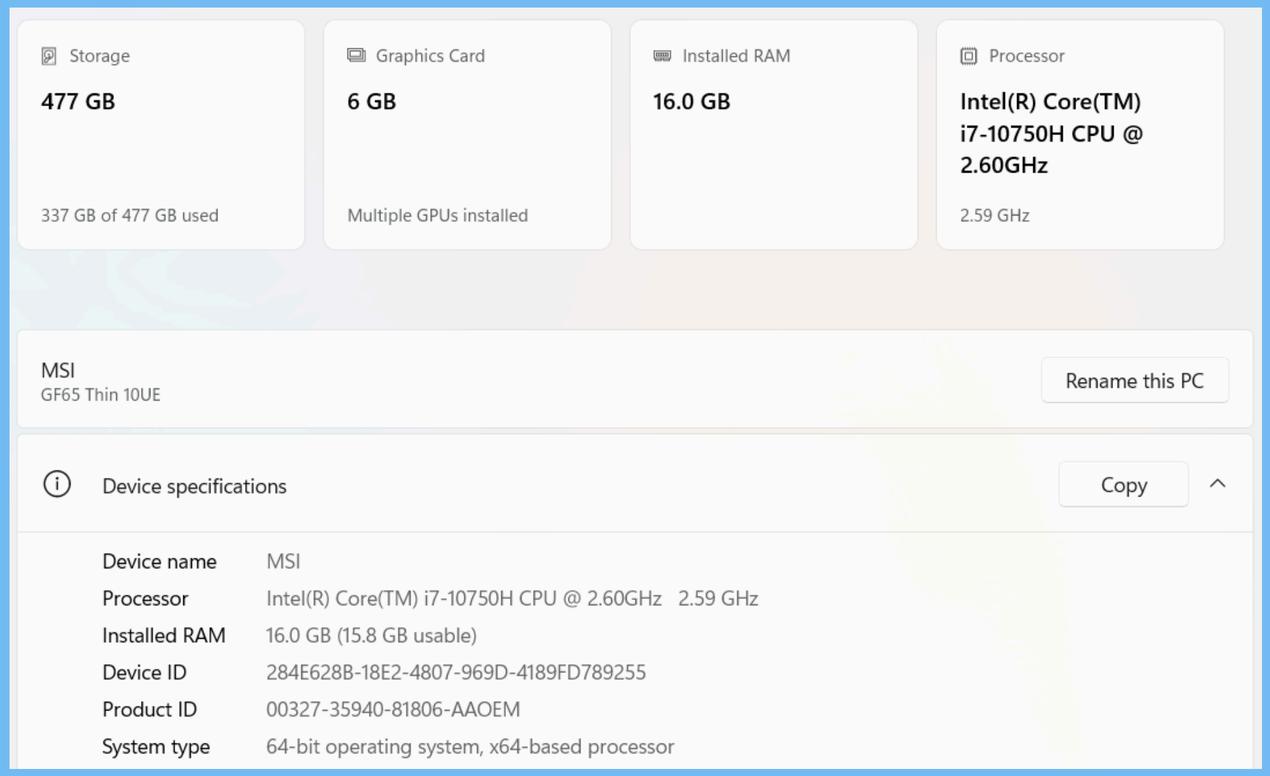
DEVICE SPECS

1211107904 - Yeong Zi Yan



DEVICE SPECS

1211108404 - Low Wan Jin



LANGUAGE

Java

Python

Theoretical Analysis

Merge Sort

Step-by-step Logic	1. Divide the array into halves recursively until single-element arrays. 2. Merge the arrays back together in sorted order.
Time Complexity	• Best: O(n log n) • Average: O(n log n) • Worst: O(n log n)
Space Complexity	O(n)

📝 **Stable & consistent performance across all cases.**

Quick Sort

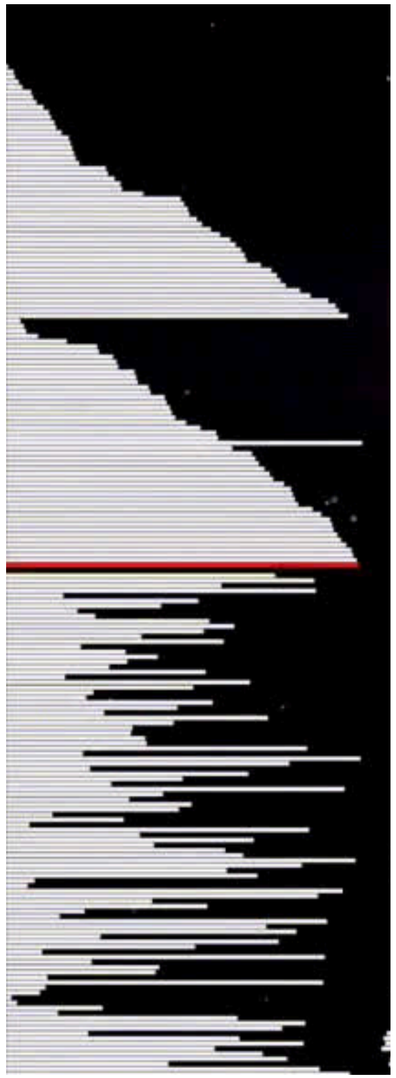
Step-by-step Logic	1. Last element as the pivot. 2. Partition the array: elements < pivot go left, > pivot go right. 3. Recursively apply to subarrays.
Time Complexity	• Best: O(n log n) • Average: O(n log n) • Worst: $O(n^2)$
Space Complexity	O(log n) (due to recursion stack)

📝 **Fast in practice, but sensitive to pivot choice.**

Merge Sort

Step-by-step Logic	1. Divide the array into halves recursively until single-element arrays. 2. Merge the arrays back together in sorted order.
Time Complexity	• Best: O(n log n) • Average: O(n log n) • Worst: O(n log n)
Space Complexity	O(n)

📝 **Stable & consistent performance across all cases.**



Quick Sort

Step-by-step Logic	1. Last element as the pivot. 2. Partition the array: elements < pivot go left, > pivot go right. 3. Recursively apply to subarrays.
Time Complexity	• Best: O(n log n) • Average: O(n log n) • Worst: $O(n^2)$
Space Complexity	O(log n) (due to recursion stack)

📝 **Fast in practice, but sensitive to pivot choice.**

Merge Sort

Step-by-step Logic	1. Divide the array into halves recursively until single-element arrays. 2. Merge the arrays back together in sorted order.
Time Complexity	• Best: O(n log n) • Average: O(n log n) • Worst: O(n log n)
Space Complexity	O(n)

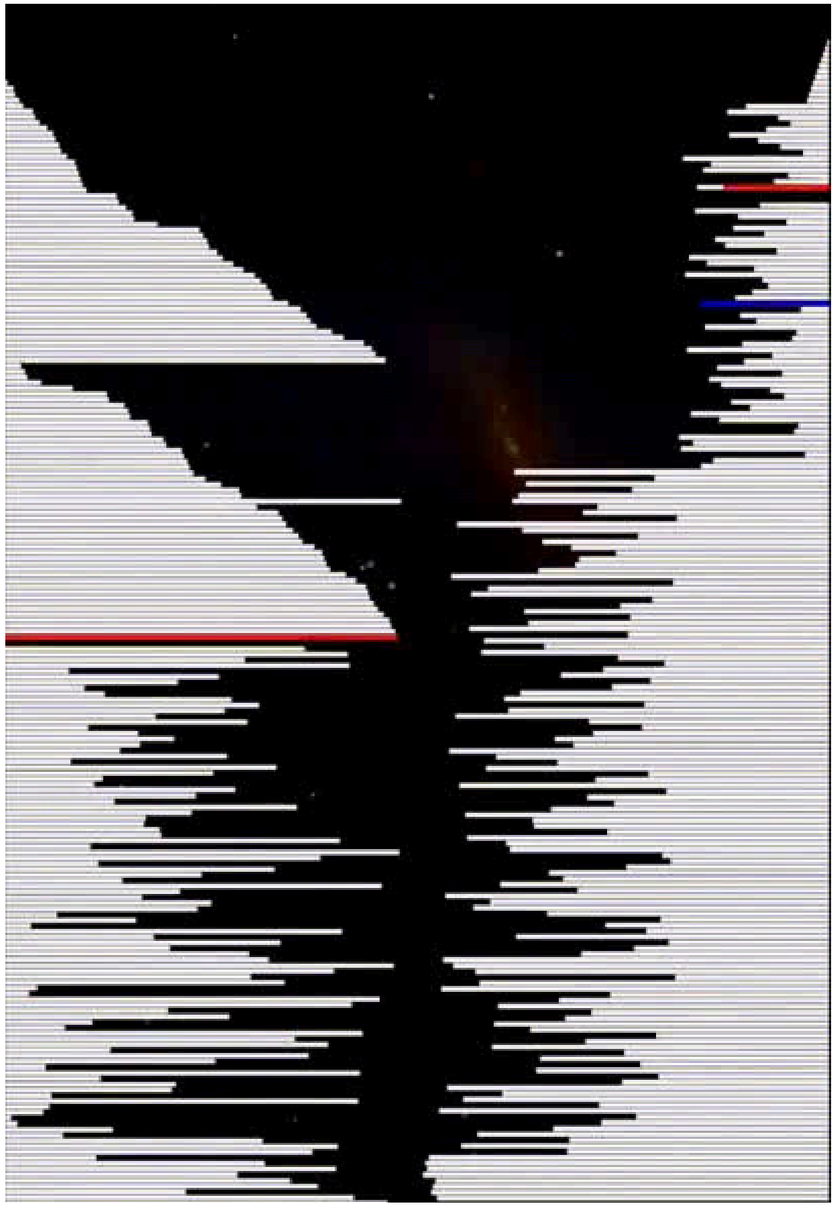
📝 **Stable & consistent performance across all cases.**

Quick Sort

Step-by-step Logic	1. Last element as the pivot. 2. Partition the array: elements < pivot go left, > pivot go right. 3. Recursively apply to subarrays.
Time Complexity	• Best: O(n log n) • Average: O(n log n) • Worst: $O(n^2)$
Space Complexity	O(log n) (due to recursion stack)

📝 **Fast in practice, but sensitive to pivot choice.**

Merge Sort



Quick Sort

Binary Search

Search Process	1. Compare middle element with target. 2. If equal, return. 3. If target < middle, search left; else, search right. 4. Repeat until found or subarray is empty.
Time Complexities	• Best Case: Target is at the middle → O(1) • Average Case: Target is somewhere in the array → O(log n) • Worst Case: Target is at one end or not found → O(log n)
Space Complexity	• Iterative: O(1) • Recursive: O(log n)

📝 **Only works on sorted arrays.**

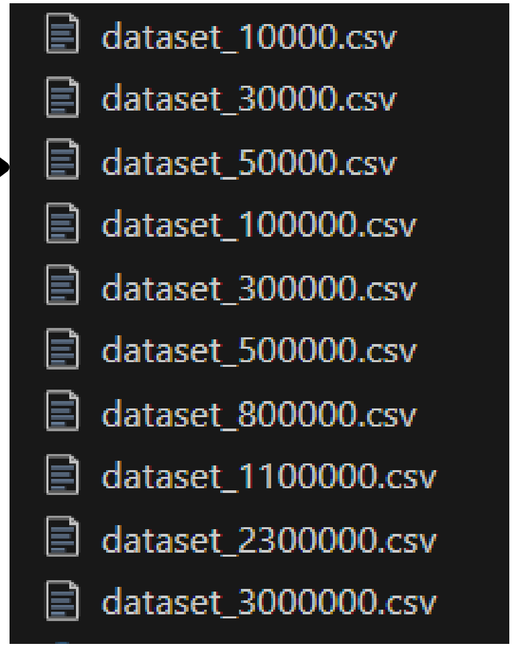
Experimental Study (Runtime Analysis)

Tang Wei Xong

Dataset Summary

Dataset format:
(integer, string)

Sizes used:
10000 - 3 million



Example running time:

dataset_10000.csv	D:\1. MMU\Year 2\Algorithm\Assigment>java quick_sort.java Quick Sort runtime for dataset size 10000: 0.017 seconds D:\1. MMU\Year 2\Algorithm\Assigment>java quick_sort.java Quick Sort runtime for dataset size 10000: 0.016 seconds D:\1. MMU\Year 2\Algorithm\Assigment>java quick_sort.java Quick Sort runtime for dataset size 10000: 0.018 seconds 1st attempt: 0.017 2nd attempt: 0.016 3rd attempt: 0.018
dataset_30000.csv	D:\1. MMU\Year 2\Algorithm\Assigment>java quick_sort.java Quick Sort runtime for dataset size 30000: 0.049 seconds D:\1. MMU\Year 2\Algorithm\Assigment>java quick_sort.java Quick Sort runtime for dataset size 30000: 0.047 seconds D:\1. MMU\Year 2\Algorithm\Assigment>java quick_sort.java Quick Sort runtime for dataset size 30000: 0.042 seconds 1st attempt: 0.049 2nd attempt: 0.047 3rd attempt: 0.042

Link for running time screenshot:
https://docs.google.com/document/d/1ZmtmvPggbOK6mpp-7bnD-qIny8RORWeB5Vt1gyGvZqY/edit?usp=sharing

(each run 3 times and the average taken to ensure accuracy)

Tang Wei Xong

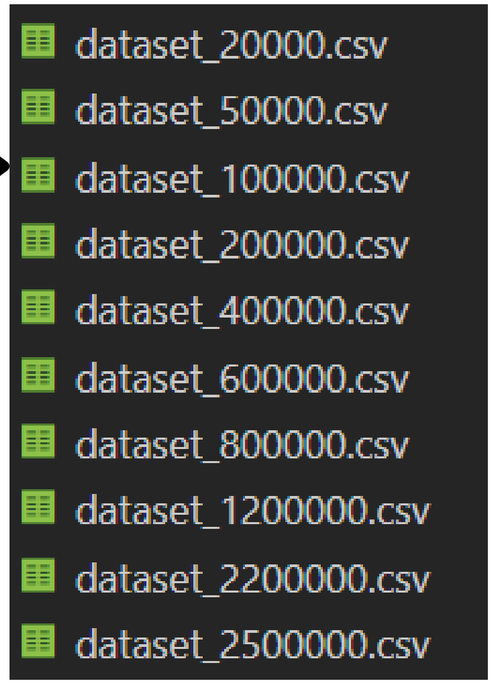
Click for Runtime Analysis

Joey Tan

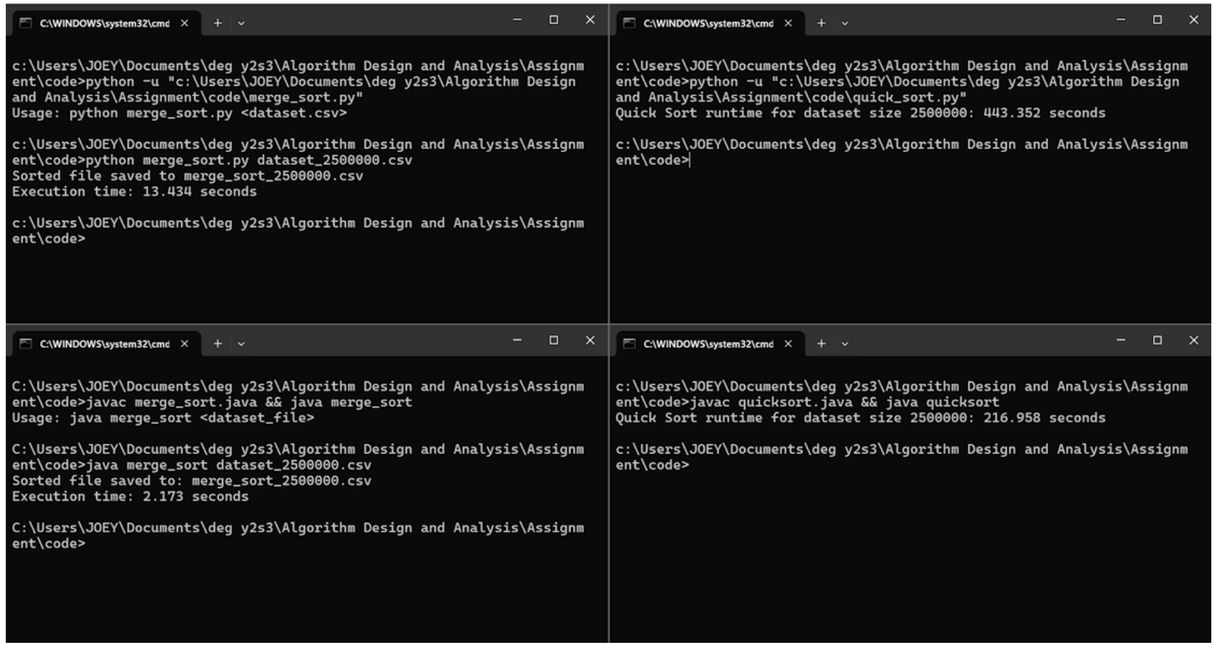
Dataset Summary

Dataset format:
(integer, string)

Sizes used:
20000 - 2.5million



Example running time:



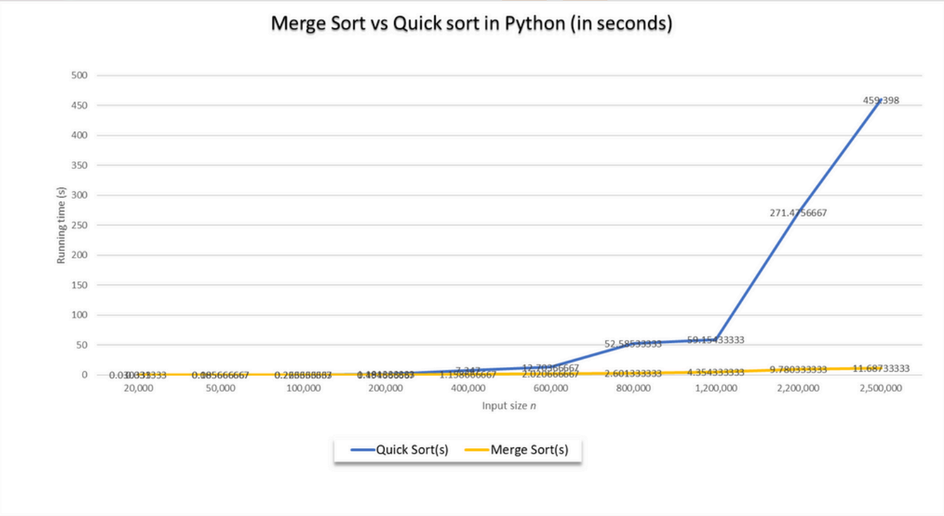
Link for running time screenshot :
https://docs.google.com/document/d/1czsY8Bse3y07fGQHPUP53VkMzyN-H0OrNLA0kBnRWGE/edit?usp=sharing

(each run 3 times and the average taken to ensure accuracy)

Joey Tan

[Excel link: Experiment study - Joey.xlsx]

Charts/Tables for Merge Sort vs Quick sort in **Python** (in seconds):



Merge Sort vs Quick sort in Python (in seconds)			
	Running time (s)		
Input size n	Quick Sort(s)	Merge Sort(s)	Time gaps
20,000	0.030333333	0.031	-0.000666667
50,000	0.105666667	0.085666667	0.02
100,000	0.266666667	0.223333333	0.043333333
200,000	1.181333333	0.494666667	0.686666667
400,000	7.347	1.158666667	6.188333333
600,000	12.70366667	2.020666667	10.683
800,000	52.58533333	2.601333333	49.984
1,200,000	59.15433333	4.354333333	54.8
2,200,000	271.4756667	9.780333333	261.6953333
2,500,000	459.398	11.68733333	447.7106667

The provided table compares Python implementations of Merge Sort and Quick Sort across various input sizes, with each run conducted three times and the average taken to ensure accuracy.

- While **both are fast** for small datasets, Merge Sort consistently outperforms Quick Sort as data grows.

This aligns with **theoretical complexities:**
Merge Sort maintains **O(nlogn)**, whereas Quick Sort, with its last-element pivot, can degrade toward **O(n²)**.

For example:

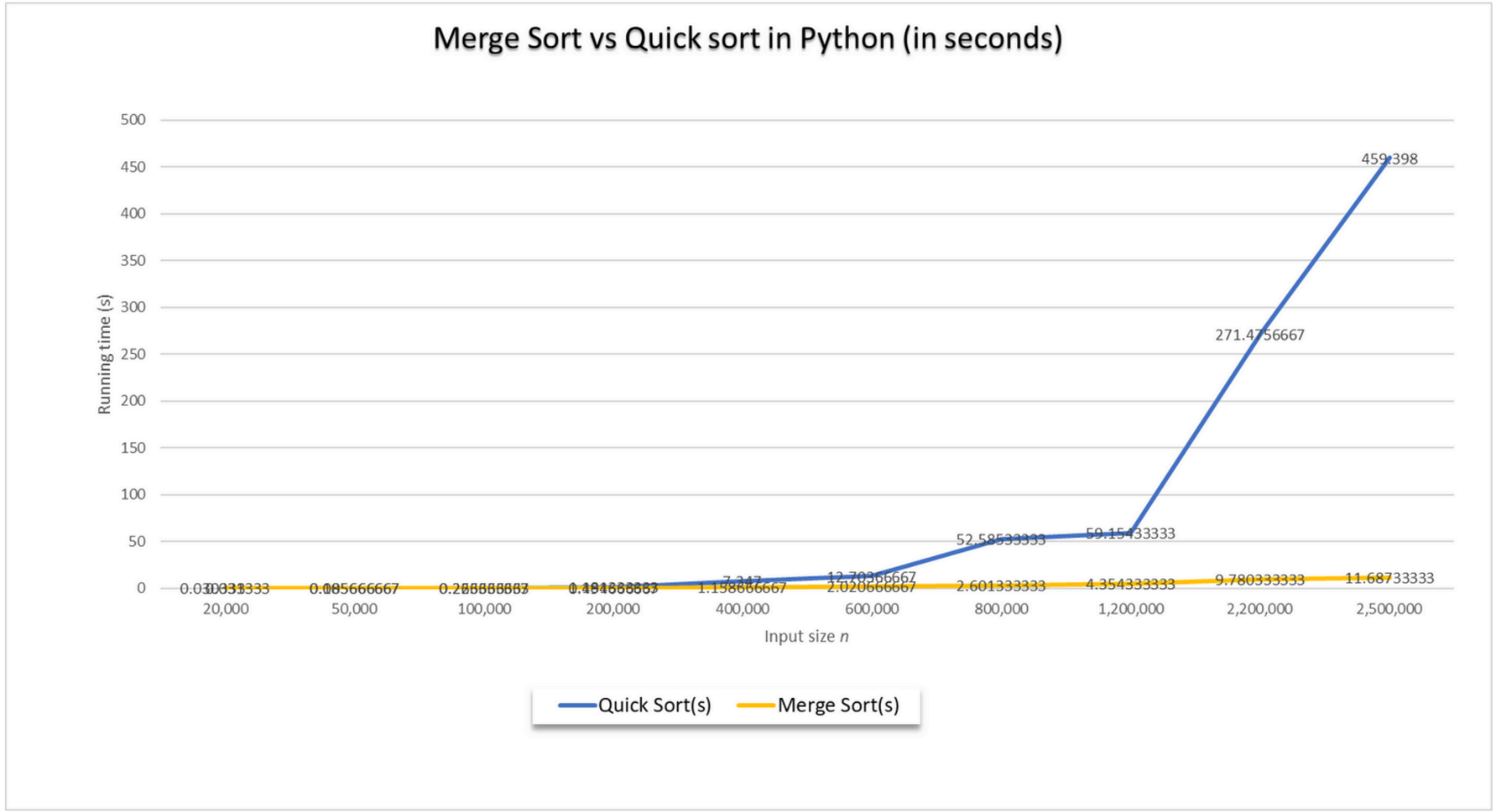
- **800,000 elements:**
  - Quick Sort : 52.58 seconds
  - Merge Sort's 2.60 seconds.
- **2,500,000 elements:**
  - Quick Sort expanding to 459.39 seconds
  - Merge Sort completes in just 11.68 seconds.
- Crucially, the time gap for the largest dataset is well over the required 60 seconds.

This data strongly suggests Merge Sort is more efficient and scales better for large datasets in this specific implementation, consistent with theoretical expectations.

Joey Tan

[Excel link: Experiment study - Joey.xlsx]

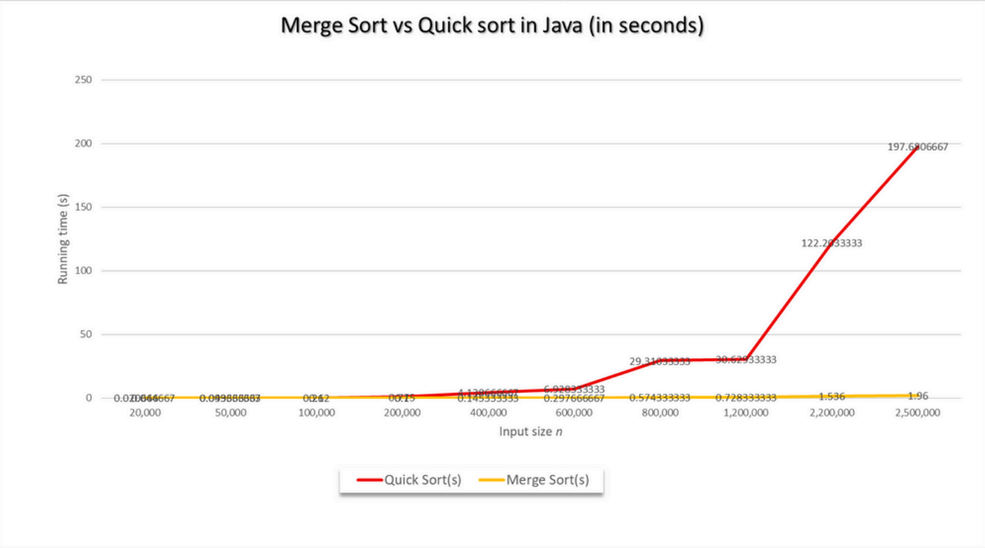
Chart for Merge Sort vs Quick sort in **Python** (in seconds):



Joey Tan

[Excel link: Experiment study - Joey.xlsx]

Charts/Tables for Merge Sort vs Quick sort in **Java** (in seconds):



Merge Sort vs Quick sort in Java (in seconds)			
	Running time (s)		
Input size n	Quick Sort(s)	Merge Sort(s)	Time gaps
20,000	0.044	0.020666667	0.023333333
50,000	0.099666667	0.043333333	0.056333333
100,000	0.262	0.1	0.162
200,000	0.775	0.1	0.675
400,000	4.138666667	0.145333333	3.993333333
600,000	6.928333333	0.297666667	6.630666667
800,000	29.31033333	0.574333333	28.736
1,200,000	30.62933333	0.728333333	29.901
2,200,000	122.2033333	1.536	120.6673333
2,500,000	197.6806667	1.96	195.7206667

The provided table compares Python implementations of Merge Sort and Quick Sort across various input sizes, with each run conducted three times and the average taken to ensure accuracy.

- While **both are fast** for small datasets, Merge Sort consistently outperforms Quick Sort as data grows.

This aligns with **theoretical complexities:**
Merge Sort maintains **O(nlogn)**, whereas Quick Sort, with its last-element pivot, can degrade toward **O(n²)**.

For example:

- **800,000 elements:**
  - Quick Sort : 29.31 seconds
  - Merge Sort's 0.57 seconds.
- **2,500,000 elements:**
  - Quick Sort expanding to 197.68 seconds
  - Merge Sort completes in just 1.96 seconds.

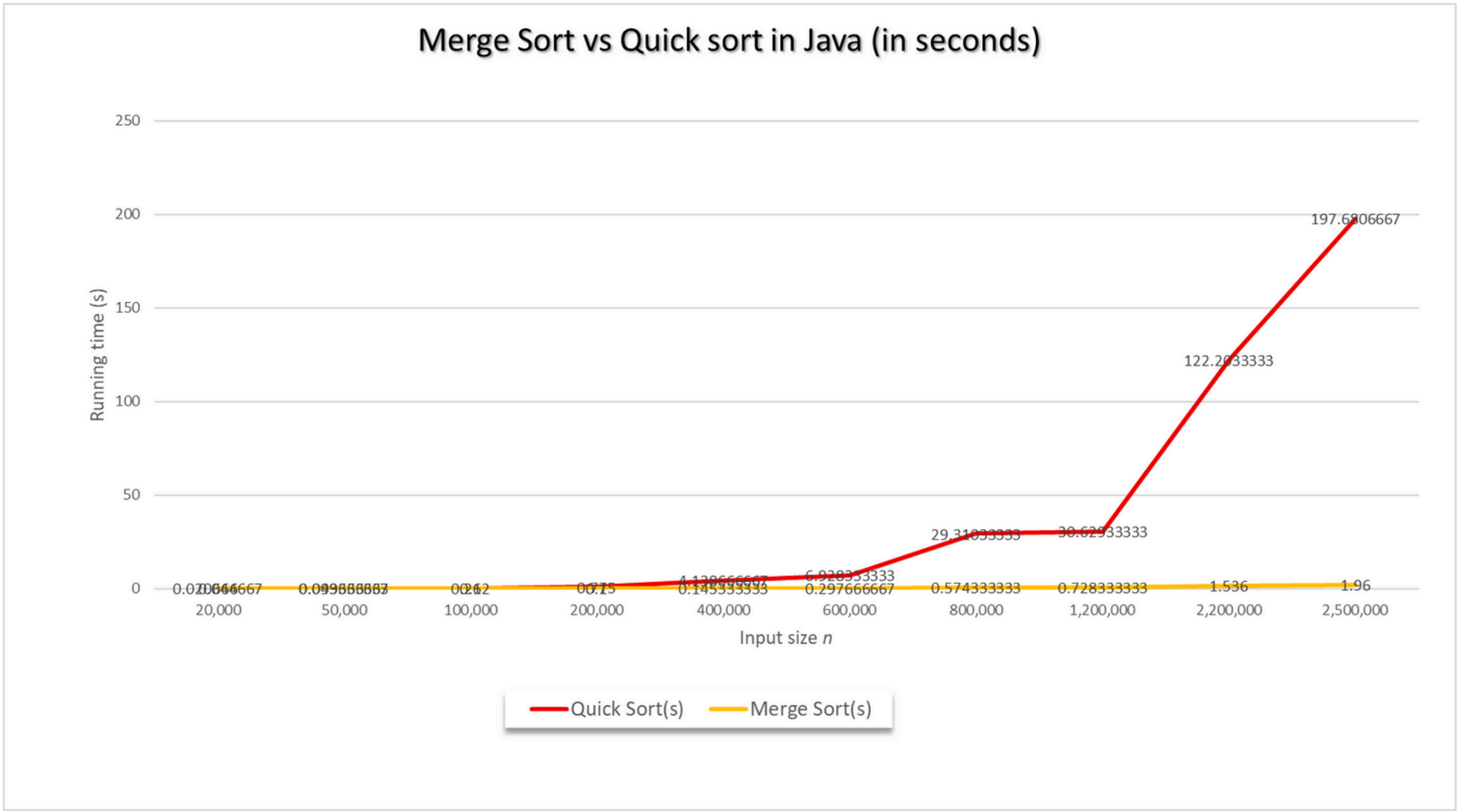
- Crucially, the time gap for the largest dataset is well over the required 60 seconds.

This data strongly suggests Merge Sort is more efficient and scales better for large datasets in this specific Java implementation, consistent with theoretical expectations.

Joey Tan

[Excel link: Experiment study - Joey.xlsx]

Charts for Merge Sort vs Quick sort in **Java** (in seconds):



Joey Tan

[Excel link: Experiment study - Joey.xlsx]

Charts/Tables for Binary Search Runtime (Best, Average, Worst Case) in Python vs Java (nanoseconds) :

Binary Search Runtime (Best, Average, Worst Case) in Python vs Java						
			Running time (ns)			
		Java			Python	
Input size n	Java - Best case (ns)	Java - Average case (ns)	Java - Worst case (ns)	Python - Best case (ns)	Python - Average case (ns)	Python - Worst case (ns)
20,000	149.835	159.0016667	140.96	1137.68	1533.063333	1151.483333
50,000	102.92	170.8906667	96.19266667	1104.04	1387.033333	1142.136667
100,000	84.75666667	185.157	71.09366667	1212.81	1743.126667	1231.83
200,000	74.61183333	314.6135	69.687	1199.49	2961.336667	1373.36
400,000	64.20416667	389.7851667	63.45441667	1297.146667	2970.76	1464.593333
600,000	64.89	443.9001667	68.46072233	1407.166667	2998.92	1474.886667
800,000	62.087875	508.7894583	63.76520833	1432.106667	3295.406667	1454.533333
1,200,000	61.57827767	591.591722	62.89852767	1501.993333	3777.073333	1572.443333
2,200,000	63.43209067	691.0016667	66.98796967	1586.89	4049.176667	1640.636667
2,500,000	62.22589333	713.4440933	70.35478667	1505.556667	4206.09	1558.863333

The graph and table compare Binary Search runtimes (best, average, worst cases) in Java and Python across varying input sizes, averaged over **three runs** in nanoseconds.

- Java's compiled execution consistently delivers faster results.

For example:

- **Input size 600,000** (avg case):
  - Java: ~443 ns
  - Python: ~2998 ns
- **Input size 2,500,000** (avg case):
  - Java: ~713 ns
  - Python: ~4206 ns.

As expected, both languages follow the theoretical time complexity:

- Average case & Worst cases: **O(log n)**
- Best case : **(O(1))**

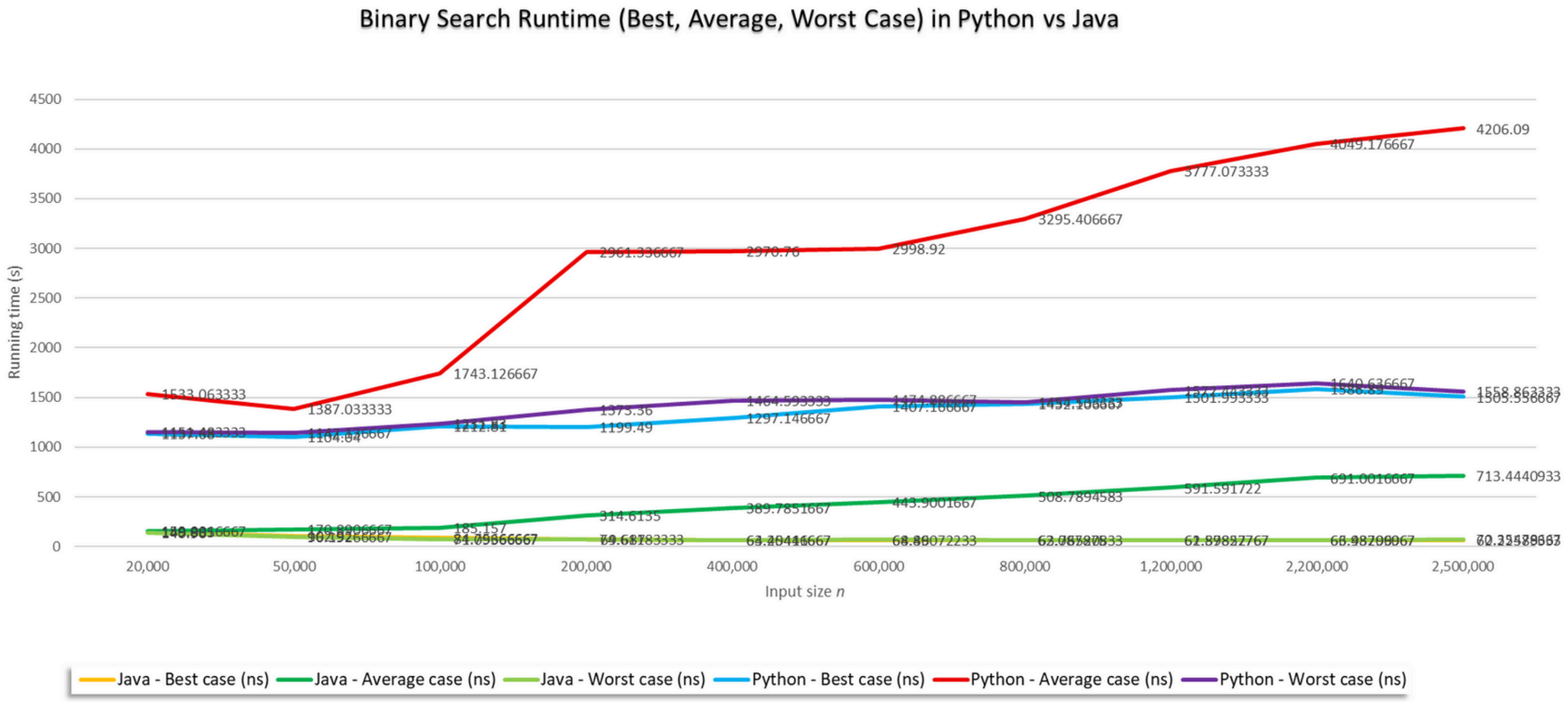
Java's best and worst case timings are often closely aligned due to minimal overhead and measurement granularity, while Python's interpreted execution introduces noticeable latency.

Overall, **Java is more efficient** for Binary Search, especially at scale, reinforcing its advantage in performance-critical applications.

Joey Tan

[Excel link: Experiment study - Joey.xlsx]

Charts for Binary Search Runtime (Best, Average, Worst Case) in Python vs Java (nanoseconds) :

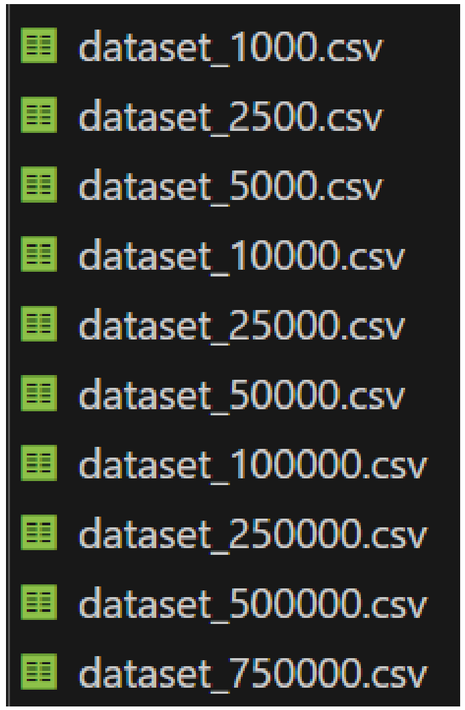


Yeong Zi Yan

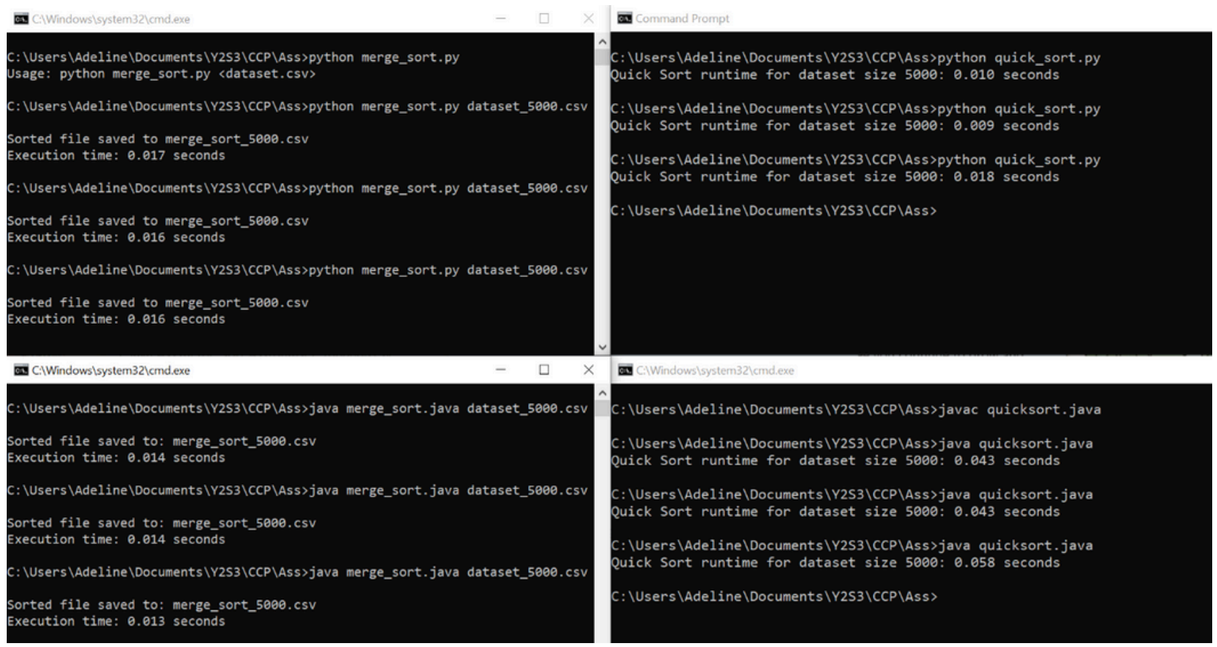
Dataset Summary

Dataset format:
(integer, string)

Sizes used:
1,000 - 750,000



Example running time:

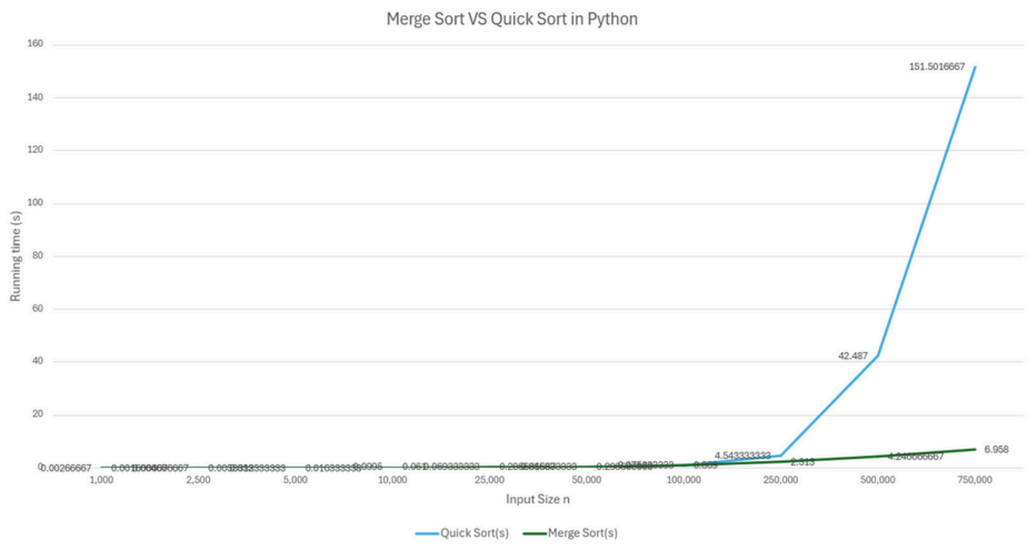


Link for running time screenshot :
https://docs.google.com/document/d/1jAJ10TxiO1zrWS7Sh0GG5L7V6WV1bGelCnBnJdL6Z3Q/edit?usp=sharing

(each run 3 times and the average taken to ensure accuracy)

Yeong Zi Yan

Charts/Tables for Merge Sort vs Quick sort in **Python** (in seconds):



Merge Sort vs Quick sort in Python (in seconds)			
	Running time (s)		
Input size n	Quick Sort(s)	Merge Sort(s)	Time gaps
1,000	0.00266667	0.001666667	0.001000003
2,500	0.004666667	0.0053333	0.000666633
5,000	0.012333333	0.016333333	-0.004
10,000	0.0995	0.061	0.0385
25,000	0.069333333	0.286666667	-0.217333334
50,000	0.315333333	0.298666666	0.016666667
100,000	0.975333333	0.869	0.106333333
250,000	4.543333333	2.313	2.23033333
500,000	42.487	4.246666667	38.24033333
750,000	151.5016667	6.958	144.5436667

This section presents a performance comparison between Merge Sort and Quick Sort implemented in Python. Each dataset size was executed three times, and the average runtime was taken to ensure consistency.

- Datasets ranged from 1,000 to 750,000
- Both algorithms delivered fast results on smaller inputs (up to 100,000 records)

This aligns with their time complexities: Merge Sort keeps a stable O(n log n), while Quick Sort, using a last-element pivot, can degrade to $O(n^2)$.

At 500,000 records:

- Quick Sort: 42.49 seconds
- Merge Sort: 4.25 seconds

At 750,000 records:

Quick Sort: 151.50 seconds

Merge Sort: 6.96 seconds

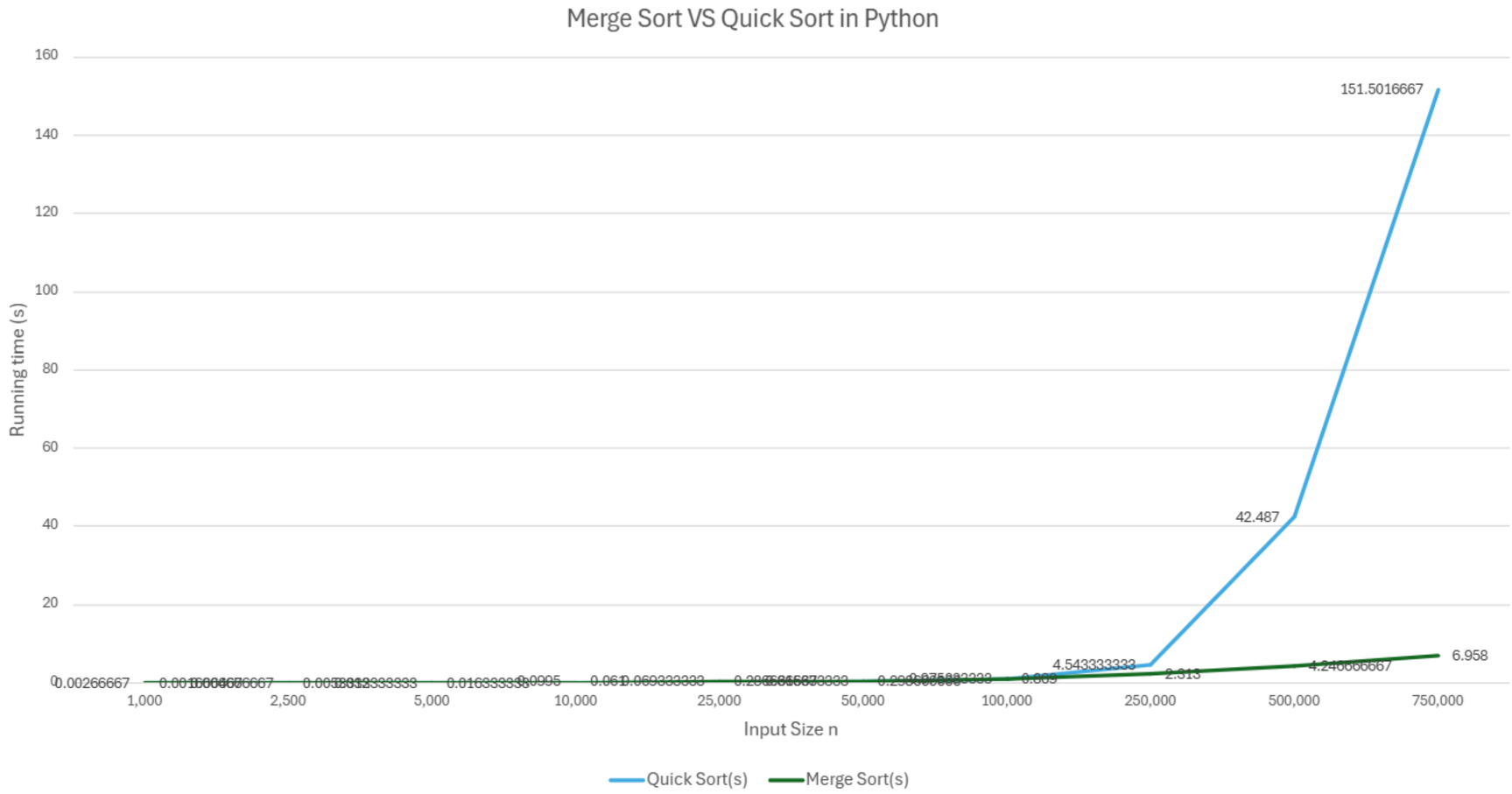
These results reveal a widening efficiency gap as data volume increases. Merge Sort maintains reliable speed, while Quick Sort slows significantly due to less balanced divisions in larger datasets.

- Merge Sort proves to be more reliable and scalable for large datasets in Python
- Quick Sort may perform acceptably for smaller sizes but struggles under heavy load
- For data-intensive Python applications, selecting a stable sorting algorithm like Merge Sort can result in substantial time savings

Yeong Zi Yan

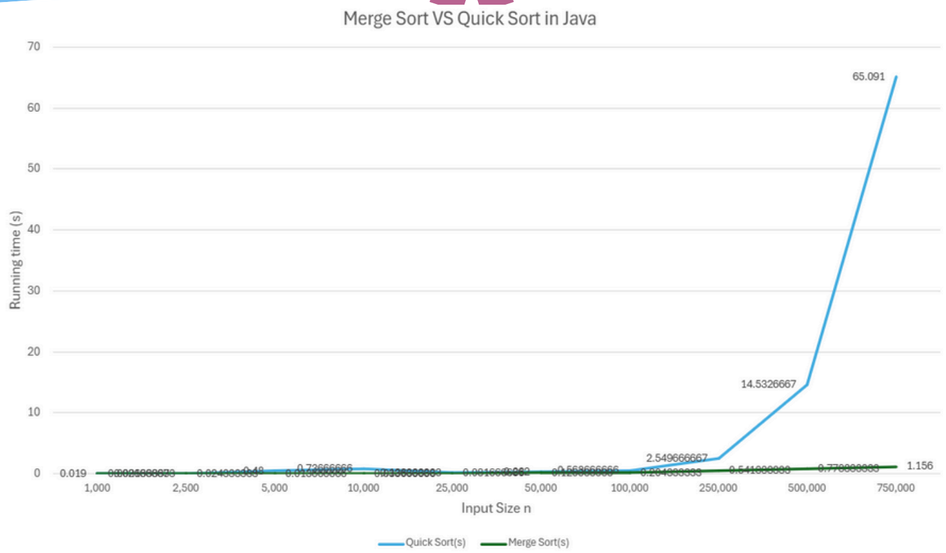
[Excel link: Experiment study - YeongZiYan]

Chart for Merge Sort vs Quick sort in **Python** (in seconds):



Yeong Zi Yan

Charts/Tables for Merge Sort vs Quick sort in **Java** (in seconds):



Merge Sort vs Quick sort in Java (in seconds)			
	Running time (s)		
Input size n	Quick Sort(s)	Merge Sort(s)	Time gaps
1,000	0.019	0.002666667	0.016333333
2,500	0.031333333	0.024333333	0.007
5,000	0.48	0.013666666	0.466333334
10,000	0.73666666	0.033666666	0.702999994
25,000	0.136333333	0.081666666	0.054666667
50,000	0.262	0.125333333	0.136666667
100,000	0.563666666	0.204333333	0.359333333
250,000	2.549666667	0.541333333	2.008333334
500,000	14.5326667	0.778333333	13.75433337
750,000	65.091	1.156	63.935

The table and graph below illustrate the performance of Merge Sort and Quick Sort algorithms implemented in Java. Each test was run three times, and the average time was recorded for accuracy.

- Input sizes ranged from 1,000 to 750,000 elements
- Both algorithms perform well on small datasets (eg. 25,000 elements and below)

This trend is supported by their underlying time complexities:

- Merge Sort has a stable time complexity of O(n log n).
- Quick Sort, using a last-element pivot, can degrade to $O(n^2)$ in the worst case.

At 500,000 elements:

- Quick Sort = 14.53 s
- Merge Sort = 0.78 s

At 750,000 elements:

- Quick Sort = 65.09 s
- Merge Sort = 1.16 s

The performance gap widens as the input grows. Quick Sort becomes slower due to unbalanced partitions, while Merge Sort handles the increasing load efficiently with consistent divide-and-conquer logic.

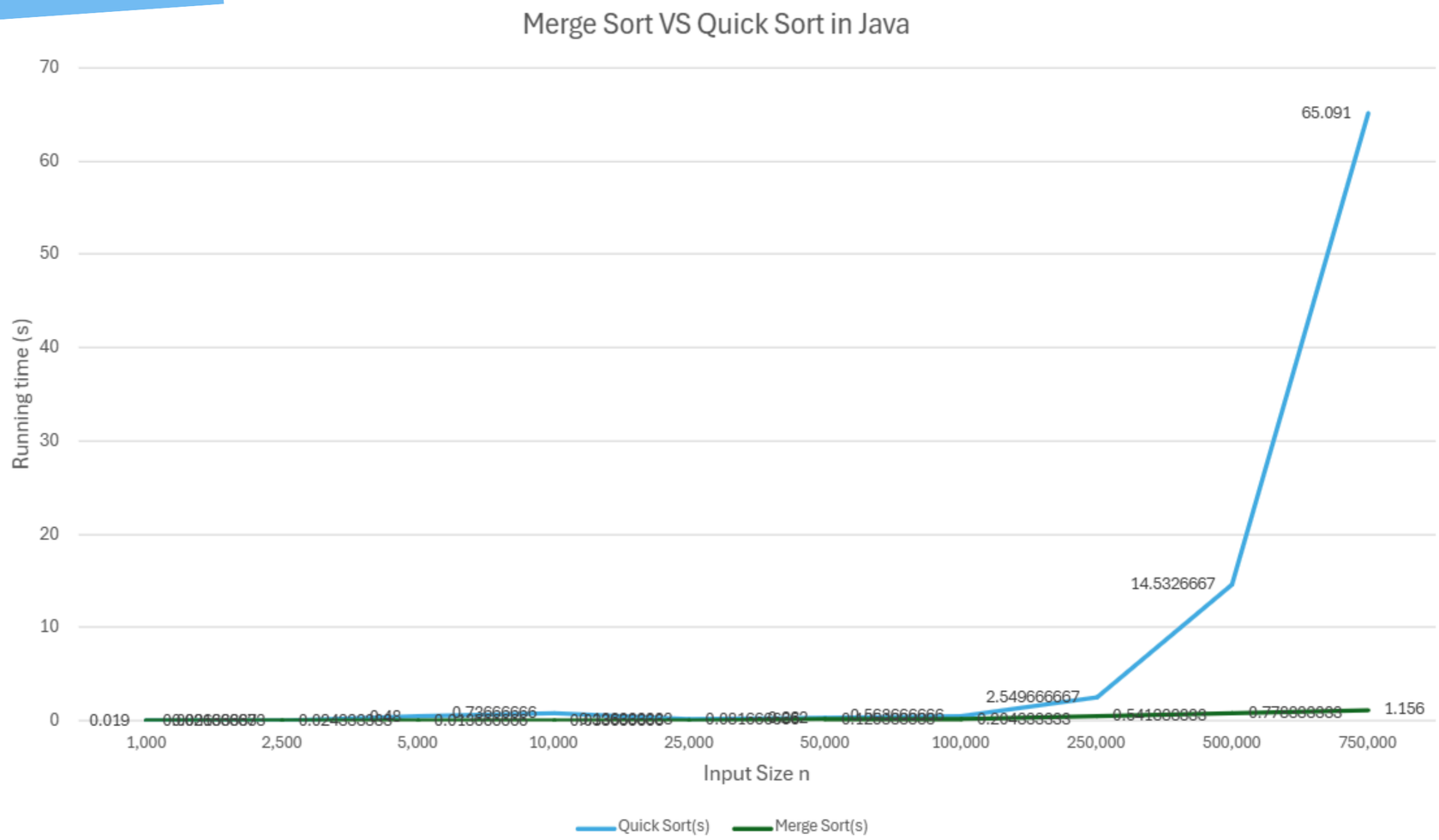
Conclusion

- Merge Sort is more scalable and efficient in this Java implementation.
- Quick Sort may work well for small data, but becomes inefficient as size increases.
- Choosing the right algorithm matters greatly when working with large datasets.

Yeong Zi Yan

[Excel link: Experiment study - YeongZiYan]

Charts for Merge Sort vs Quick sort in **Java** (in seconds):



Yeong Zi Yan

[Excel link: Experiment study - YeongZiYan]

Charts/Tables for Binary Search Runtime (Best, Average, Worst Case) in Python vs Java (nanoseconds) :

Binary Search Runtime (Best, Average, Worst Case) in Python vs Java						
	Running time (ns)					
		Java			Python	
Input size n	Java - Best case (ns)	Java - Average case (ns)	Java - Worst case (ns)	Python - Best case (ns)	Python - Average case (ns)	Python - Worst case (ns)
1,000	776.57	454.47	335.67	1423.23	1494.6	1803.87
2,500	488.6	751.69	409.05	1793.33	2494.77	3000.96
5,000	836.8	513.92	654.93	2618.16	2657.9	3117.45
10,000	828.48	427.26	709.21	2846.72	2656.18	2428.13
25,000	426.015	316.88	286.8	3165.85	3437.72	3218.15
50,000	346.07	578.994	366.27	2981.18	4306.53	3152.79
100,000	213.23	612.47	215.78	3122.67	4476.44	3545.46
250,000	147.36	763.84	136.2	3696.13	4973.68	3712.61
500,000	137.68	863.52	122.78	4187.56	5450.07	4375.18
750,000	1576.62	1044.05	160.47	4428.6	5979.58	4439.65

This comparison shows Binary Search runtimes in Java and Python across input sizes from 1,000 to 750,000 elements. Each test was run 3 times, and the average time (in nanoseconds) was recorded.

- Java consistently outperforms Python in all scenarios.

At 100,000 elements (average case):

- Java: 612 ns
- Python: 4,476 ns

At 750,000 elements (average case):

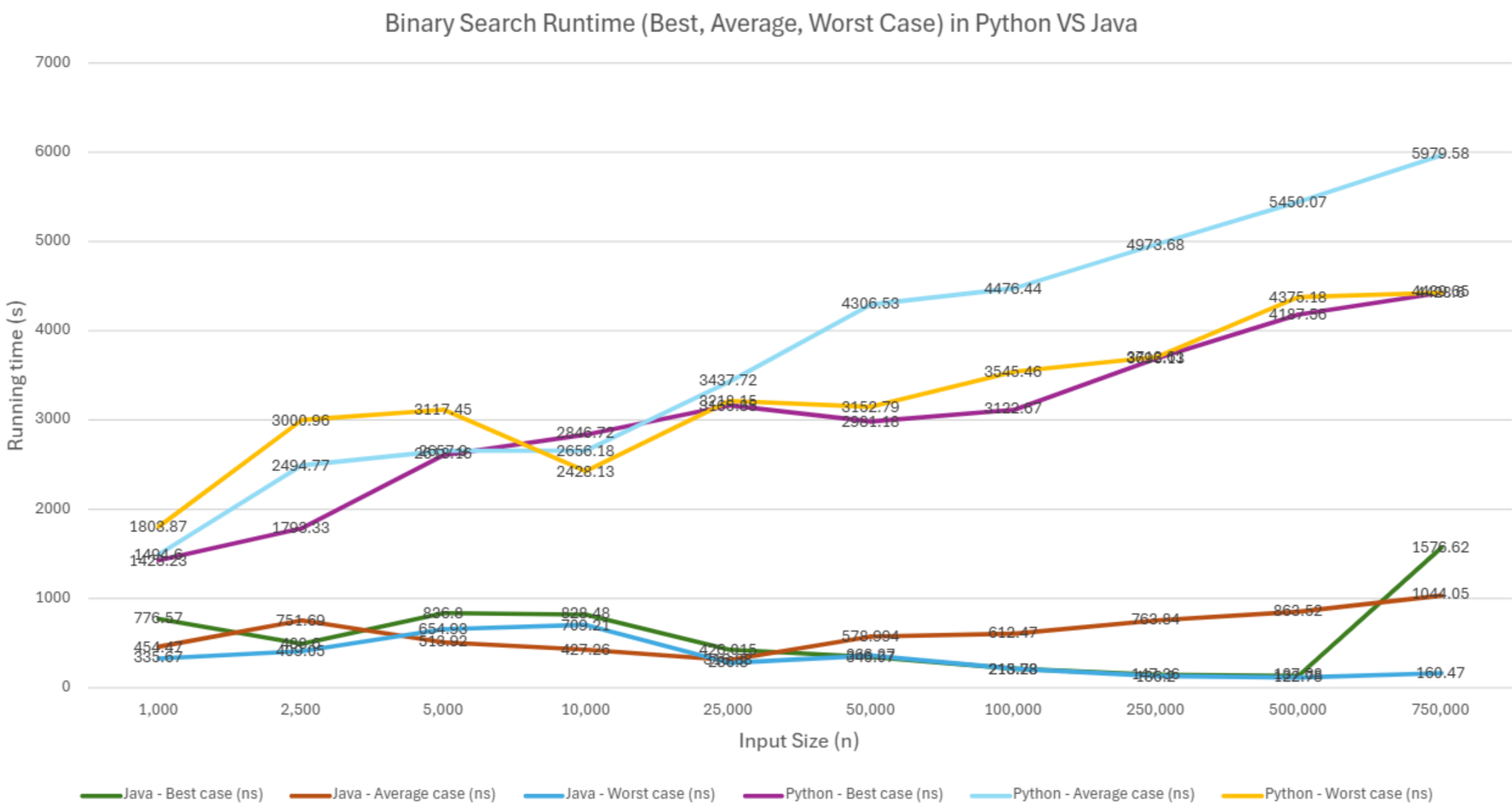
- Java: 1,044 ns
- Python: 5,979 ns

Java's compiled execution is faster and more stable for Binary Search across all input sizes as, best and worst cases are often close due to low overhead. Python shows more fluctuation due to interpreter latency and dynamic typing, especially in large datasets. While both follow O(log n) complexity, Java is better suited for performance-critical or large-scale binary search tasks whereas Python suits simpler or less time-sensitive tasks.

Yeong Zi Yan

[Excel link: Experiment study - Joey.xlsx]

Charts for Binary Search Runtime (Best, Average, Worst Case) in Python vs Java (nanoseconds) :

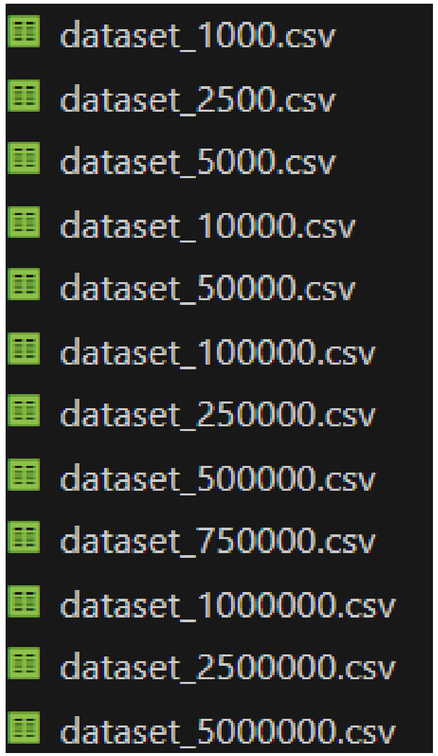


Low Wan Jin

Dataset Summary

Dataset format:
(integer, string)

Sizes used:
1000 - 5million



Example running time:

```
C:\Users\USER\Documents\algorithm design and analysis\assignment\assignment\src>java merge_sort dataset_5000000.csv
Sorted file saved to: merge_sort_5000000.csv
Execution time: 4.115 seconds


C:\Users\USER\Documents\algorithm design and analysis\assignment\assignment\src>python merge_sort.py dataset_5000000.csv
Sorted file saved to merge_sort_5000000.csv
Execution time: 27.919 seconds


C:\Users\USER\Documents\algorithm design and analysis\assignment\assignment\src>javac quicksort.java && java quicksort
Quick Sort runtime for dataset size 5000000: 629.892 seconds


C:\Users\USER\Documents\algorithm design and analysis\assignment\assignment\src>python -u "C:\Users\USER\Documents\algorithm design and analysis
\assignment\assignment\src\quick_sort.py"
Quick Sort runtime for dataset size 5000000: 1387.733 seconds
```

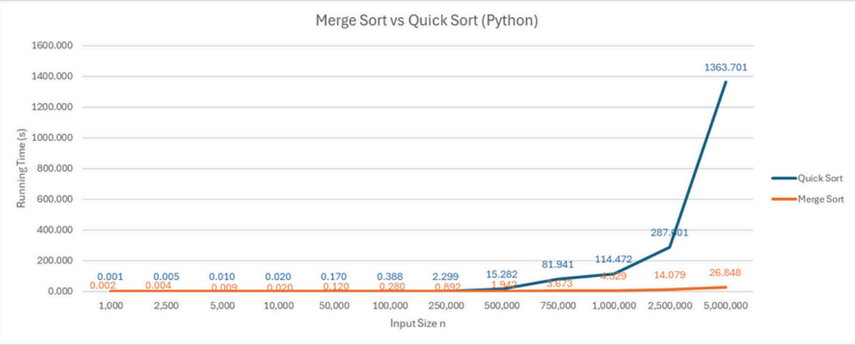
Link for running time screenshot :
https://docs.google.com/document/d/1ld_h9-AAiOnJihaeB2y6XdE0i-h7UCnW_ByxEsIJtzs/edit?usp=sharing

(each run 3 times and the average taken to ensure accuracy)

Low Wan Jin

Excel link: Experimental Study - Wan Jin

Merge Sort vs Quick Sort (Python)



Input size n	Average Running Time (s)		
	Python		
	Quick Sort	Merge Sort	Time difference (python)
1,000	0.001	0.002	0.000
2,500	0.005	0.004	0.001
5,000	0.010	0.009	0.001
10,000	0.020	0.020	0.001
50,000	0.170	0.120	0.050
100,000	0.388	0.280	0.108
250,000	2.299	0.892	1.407
500,000	15.282	1.942	13.340
750,000	81.941	3.673	78.268
1,000,000	114.472	4.329	110.142
2,500,000	287.601	14.079	273.522
5,000,000	1363.701	26.848	1336.853

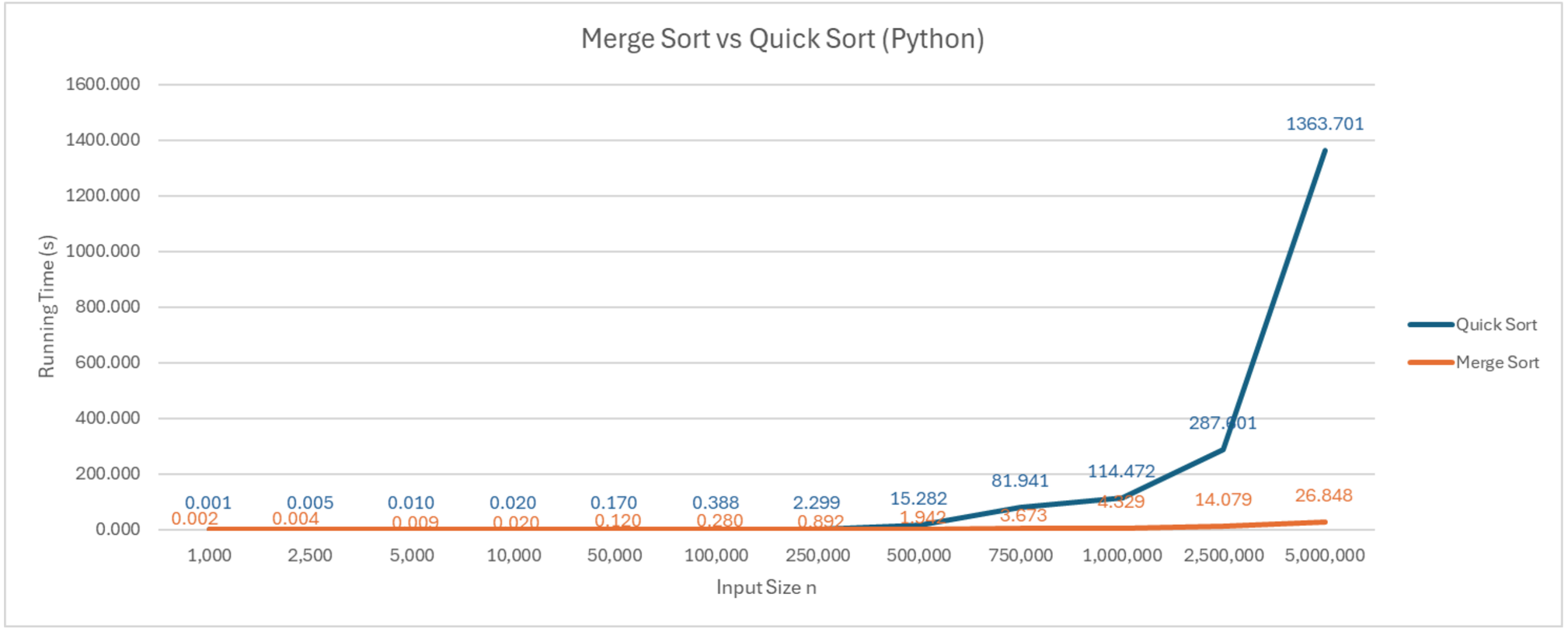
Graph Description: The graph visually reinforces the table's findings, plotting the running time (in seconds) against the input size for Merge Sort and Quick Sort in Python. The Merge Sort line remains relatively flat and close to the x-axis, illustrating its efficient and stable performance (O(nlogn)). In stark contrast, the Quick Sort line shows a dramatic, almost vertical increase for larger input sizes, indicative of a less efficient scaling behavior, potentially approaching $O(n^2)$ in this Python implementation.

Table Description: The table presents the average running times (in seconds) for Quick Sort and Merge Sort across input sizes ranging from 1,000 to 5,000,000. It highlights that while both are very fast for small inputs, Merge Sort consistently demonstrates significantly faster execution times than Quick Sort as the input size increases, particularly beyond 250,000 elements.
For example, at an input size of 5,000,000, Quick Sort took 1363.701 seconds, whereas Merge Sort completed in just 26.848 seconds, highlighting Quick Sort's much steeper performance degradation.

Low Wan Jin

Excel link: Experimental Study - Wan Jin

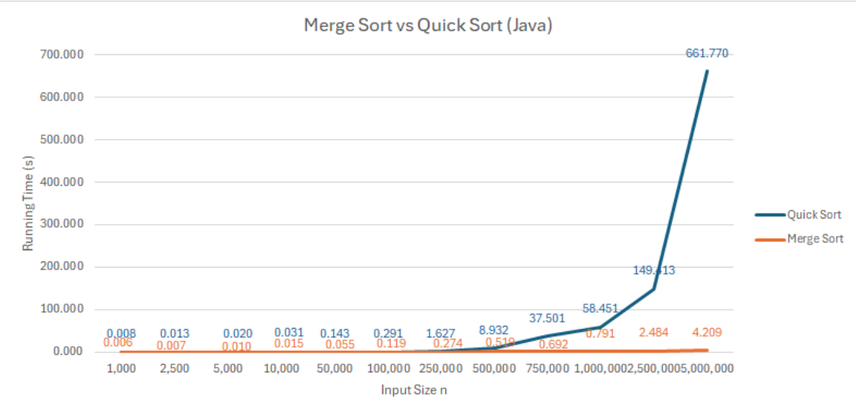
Chart for Merge Sort vs Quick Sort (Python)



Low Wan Jin

Excel link: Experimental Study - Wan Jin

Merge Sort vs Quick Sort (Java)



Input size n	Average Running time (s) Java		
	Quick Sort	Merge Sort	Time difference (Java)
1,000	0.008	0.006	0.002
2,500	0.013	0.007	0.006
5,000	0.020	0.010	0.010
10,000	0.031	0.015	0.016
50,000	0.143	0.055	0.088
100,000	0.291	0.119	0.172
250,000	1.627	0.274	1.353
500,000	8.932	0.519	8.413
750,000	37.501	0.692	36.809
1,000,000	58.451	0.791	57.660
2,500,000	149.413	2.484	146.929
5,000,000	661.770	4.209	657.561

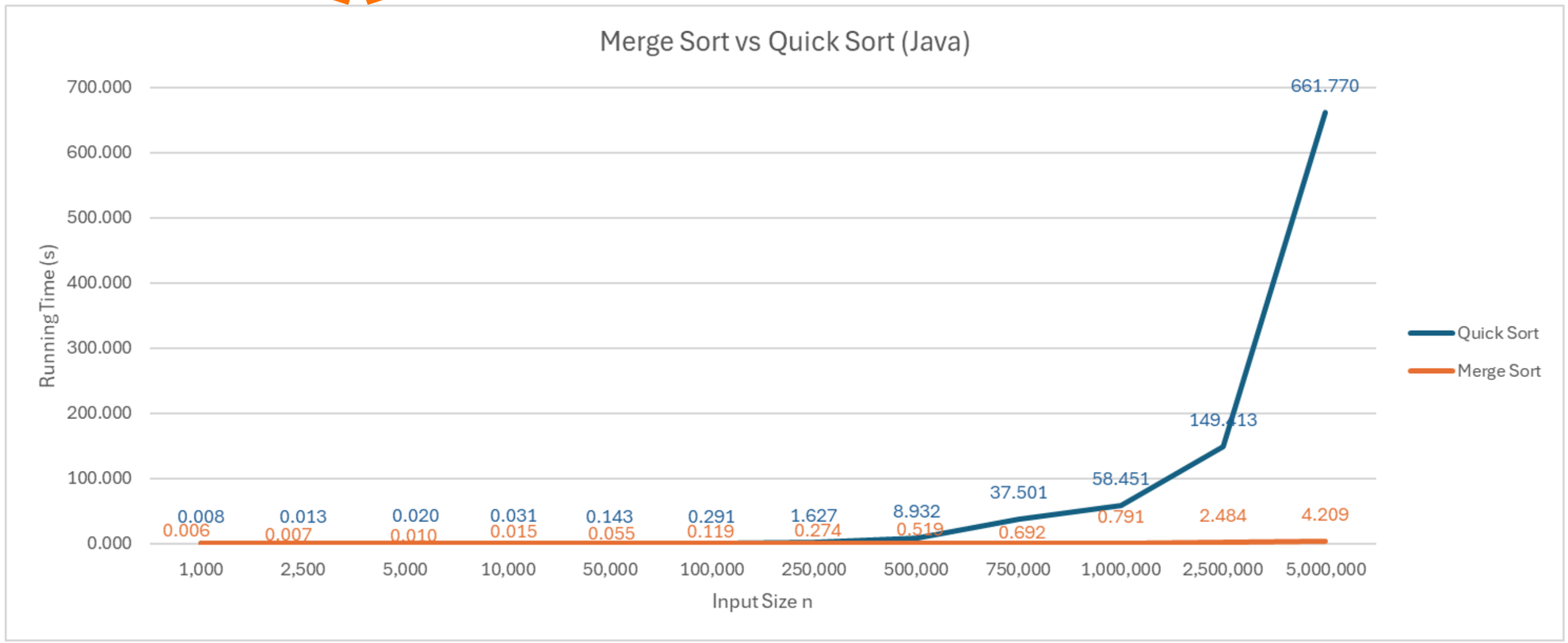
Graph Description: The graph reinforces the table's findings by illustrating the running time (in seconds) relative to input size For Merge Sort and Quick Sort in Java. The orange line, representing Merge Sort, remains consistently low and stable, reflecting its efficient time complexity of O(n log n) across various input sizes. In contrast, the blue line for Quick Sort rises sharply as the input grows, indicating a significant decline in performance. This is largely due to the use of the **last element as the pivot,** which can lead to **worst-case $O(n^2)$** behavior when the input is not well-balanced.

Table Description: The table presents the average running times (in seconds) for Quick Sort and Merge Sort in Java across input sizes ranging from 1,000 to 5,000,000. While both are very fast for small inputs, Merge Sort consistently demonstrates significantly faster execution times than Quick Sort as the input size increases. For example, at an input size of 5,000,000, Quick Sort took 1363.701 seconds, whereas Merge Sort completed in just 26.848 seconds, highlighting Quick Sort's much steeper performance degradation.

Low Wan Jin

Excel link: Experimental Study - Wan Jin

Chart for Merge Sort vs Quick Sort (Java)



Low Wan Jin

Excel link: Experimental Study - Wan Jin

Binary Search Runtime (Java vs Python in nanosecond)

Input size n	Average Running Time (ns)					
	Binary Search Java			Binary Search Python		
	Best Case	Average Case	Worst Case	Best Case	Average Case	Worst Case
1,000	542.532	266.741	204.934	1944.767	1634.333	2473.567
2,500	356.174	236.107	217.852	2624.200	2626.853	2700.253
5,000	265.527	206.875	305.928	2176.447	3251.620	2848.713
10,000	234.624	176.106	217.480	2562.287	2814.750	2987.940
50,000	119.714	189.638	141.930	2985.420	3507.840	3307.600
100,000	98.613	221.994	80.614	3466.120	3918.053	3663.733
250,000	71.397	280.506	73.388	3443.830	4436.323	3707.090
500,000	70.736	342.589	70.930	3540.287	5071.433	3926.453
750,000	62.699	395.892	69.680	3675.803	5419.310	4066.690
1,000,000	61.487	444.622	67.679	3769.303	5724.637	4155.270
2,500,000	63.839	532.702	68.662	4168.560	6633.580	4608.043
5,000,000	67.420	659.855	72.560	4352.550	6880.473	4651.030

This comparison shows Binary Search runtimes (best, average, worst cases) in Java and Python across input sizes from 1,000 to 5,000,000, averaged over three runs.

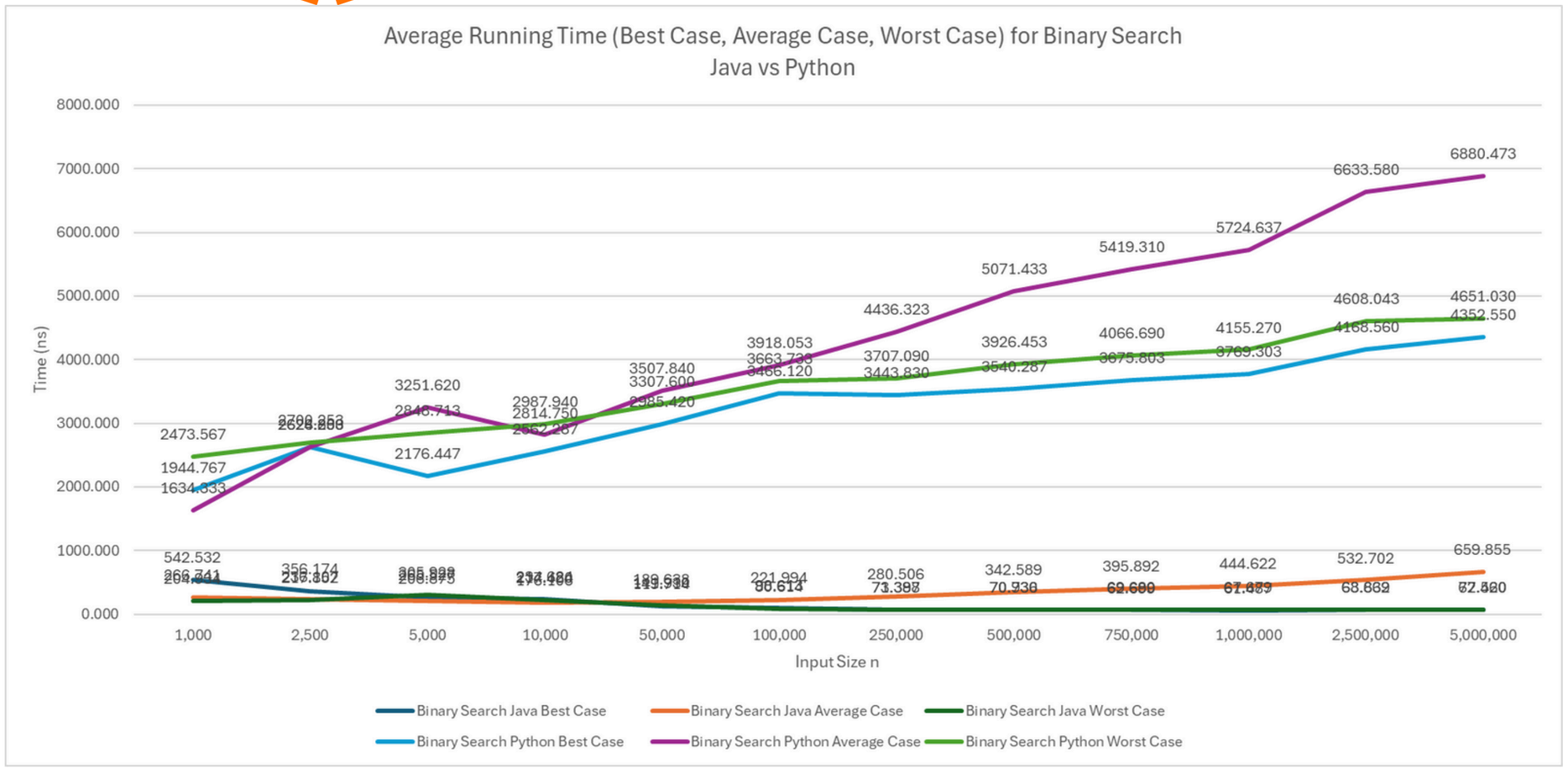
Best Case – O(1): Target is exactly in the middle, found in the first comparison.
Average Case – O(log n): Target is randomly located; search space is halved each time.
Worst Case – O(log n): Target not found or at the edge; search continues until fully narrowed down.

Java consistently performs faster due to its compiled execution. For example, at size 2,500,000, Java's average case is ~533 ns vs Python's ~6633 ns.
Both follow expected logarithmic trends, but Python shows higher latency and variability due to interpreter overhead. Java's runtimes are more stable and efficient, making it better for large-scale or performance-critical tasks.

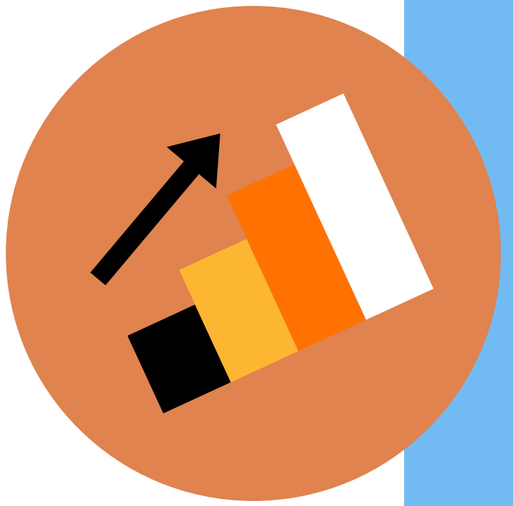
Low Wan Jin

Excel link: Experimental Study - Wan Jin

Merge Sort vs Quick Sort (Python)



Comparative Analysis

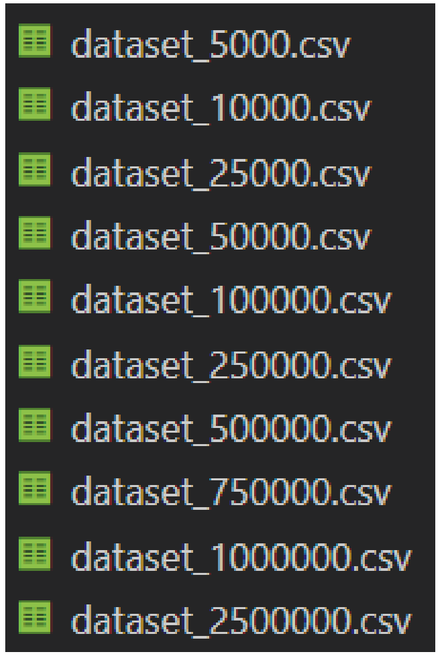


Group

Dataset Summary

Dataset format:
(integer, string)

Sizes used:
5000 - 2.5 million



Data Size	Devin				Joey				Adeline				Wanjin			
	Quick		Merge		Quick		Merge		Quick		Merge		Quick		Merge	
	Java	Py	Java	Py	Java	Py	Java	Py	Java	Py	Java	Py	Java	Py	Java	P
5000	0.013	0.008	0.01		0.007	0.01333	0.007	0.00933					0.016	0.009	0.01	
10000	0.019	0.018	0.012		0.015	0.022	0.01367	0.01367					0.025	0.021	0.015	
25000	0.052	0.056	0.021		0.044	0.05633	0.04367	0.031					0.077	0.065	0.049	
50000	0.118	0.116	0.039		0.094	0.08467	0.094	0.031					0.148	0.176	0.062	
100000	0.246	0.323	0.075		0.2916667	0.21867	0.241	0.04633					0.313	0.443	0.144	
250000	0.891	2.168	0.165		1.728	1.20333	0.67933	0.10633					1.575	2.78	0.278	
500000	4.238	11.973	0.277		11.059	6.66167	2.16267	0.44433					7.997	16.196	0.449	
750000	19.102	66.054	0.434		48.306	27.477	3.296	0.68267					39.313	83.701	0.631	
1000000	30.08	100.517	0.504		73.522	38.337	4.555	0.859					60.088	117.851	0.774	
2500000	82.101	257.244	1.601	12.487	371.82133	148.306	15.2933	2.491					151.892	295.66	2.025	1

Binary

Data Size	Devin						Joey						Adeline			
	Best		Average		Worst		Best		Average		Worst		Best		Average	
	Java	Py	Java	Py	Java	Py	Java	Py	Java	Py	Java	Py	Java	Py	Java	P
5000							284.907	1075.15	234.567	1129.05	210.467	1072				
10000							205.997	1179.06	166.34	1136.36	199.63	833.303				
25000							136.195	1034.62	168.397	1339.39	133.513	1143.75				
50000							99.364	1150.74	212.484	1519.3	98.4553	1173.64				
100000							86.9847	1167.71	216.754	2130.93	72.6117	1254.44				
250000							66.5985	1277.03	343.375	2996.04	65.1237	1397.83				
500000							62.7041	1380.57	421.779	3187.38	61.5174	1436				
750000							62.1668	1483.81	522.026	3388.67	64.4993	1576.19				
1000000							62.96	1406.98	602.777	3444.46	67.6469	1470.39				

Excel link: Experimental Study - Group

(each run 3 times and the average taken to ensure accuracy)

COMPARATIVE ANALYSIS (SORTING)

Dataset Input Size: 2,500,000 (2.5M)

Member	Runtime(s)			
	Quick Sort (Java)	Merge Sort (Java)	Quick Sort (Python)	Merge Sort (Python)
Tang Wei Xiong	82.101	1.601	257.244	12.487
Joey Tan Rou Yi	148.305	2.491	371.821	15.293
Yeong Zi Yan	406.995	6.76	972.171	29.175
Low Wan Jin	151.892	2.025	295.66	14.083

COMPARATIVE ANALYSIS (SEARCHING)

Dataset Input Size: 2,500,000 (2.5M) (SORTED)

Member	Runtime(ns)					
	Binary Search (Java)			Binary Search (Python)		
	Best Case	Average Case	Worst Case	Best Case	Average Case	Worst Case
Tang Wei Xiong	51.3496	500.8829	52.5447	2159.09	4002.03	2223.66
Joey Tan Rou Yi	64.7204	742.275	72.1908	1585.05	4512.02	1639.21
Yeong Zi Yan	74.43	885.39	76.67	4653.75	6733.86	4660.19
Low Wan Jin	65.6823	630.921	70.8721	4034.34	6331.84	4513.16

Merge Sort	**Case**	**Best Case**	**Average Case**	**Worst Case**
	Time Complexity	O (n log n)	O(n log n)	O(n log n)
	Explanation	Always divides and merges regardless of input order.	Standard divide-and-conquer on any random input.	Still consistent even with reversed input.
Quick Sort	**Case**	**Best Case**	**Average Case**	**Worst Case**
	Time Complexity	O(n log n)	O(n log n)	O(n^2)
	Explanation	When pivot divides the array evenly.	Balanced partitions occur with random data.	Worst when pivot is smallest/largest
Binary Search	**Case**	**Best Case**	**Average Case**	**Worst Case**
	Time Complexity	O(1)	O(n log n)	O(n^2)
	Explanation	Target is the middle element – found on first comparison.	Each comparison halves the array.	Even worst-case search halves till the end.

Merge Sort	**Case**	**Auxiliary Space**	**Total Space**
	Space Complexity	O(n)	O(n)
	Explanation	Needs temporary arrays to store and merge subarrays during the sorting process.	Combines the input array space (O(n)) and auxiliary space (O(n)) used for merging operations.
Quick Sort	**Case**	**Auxiliary Space**	**Total Space**
	Space Complexity	O(log n)	O(n)
	Explanation	Requires space for the recursive call stack. Balanced partitions give log n depth, worst case is linear.	Total space includes the input array (O(n)) and recursive stack space. In-place sorting uses no extra arrays.
Binary Search	**Case**	**Auxiliary Space**	**Total Space**
	Space Complexity	O(1) for iterative, O(log n) for recursive	O(n)
	Explanation	Uses constant space in iterative version (loop variables) and log n space for recursive stack.	Input array (O(n)) plus minimal extra space. Total space is dominated by the input data size.

Our Findings

a

🔍 **Same Language, Same Hardware**

- In both Java and Python, Merge Sort performed faster than Quick Sort, especially on large datasets.
- Quick Sort struggled with unbalanced partitions (due to last-element pivot).
- Binary Search had consistent fast results; the iterative version was slightly more efficient..

b

🌐 **Same Language, Different Hardware**

- Merge Sort showed better improvements on machines with more RAM (due to extra space needed).
- Quick Sort ran slightly better with higher CPU speeds, but still slower than Merge Sort overall.
- Binary Search was stable across all hardware, unless data was very large.

c

💻 **Different Language, Same Hardware**

- Java Merge Sort was faster than Python's, due to Java's compiled nature.
- In Python, Merge Sort still outperformed Quick Sort, thanks to efficient built-in list slicing.
- Binary Search was faster in Java overall, but logic remained the same.

d

💡 **Different Language, Different Hardware**

- Java on better hardware gave the best performance for Merge Sort and Binary Search.
- Python on lower-end hardware still showed Merge Sort outperforming Quick Sort, though overall speed was slower.
- Binary Search remained efficient in all setups, with minor variation.

Our Findings

e **Best sorting algorithm for the array based AVL implementation.**

Recommended Algorithm: **Merge Sort**

- **Stable:** Keeps equal elements in the same relative order—important for maintaining AVL balance on sorted insertions.
- **Predictable Time Complexity:** O(n log n) in all cases ensures consistent balancing behavior.
- **Minimized Restructuring:** Sorted output from Merge Sort reduces the chance of AVL tree becoming unbalanced during bulk insertions.
- **Handles large data well:** AVL trees benefit from merge-optimized, sorted inserts in bottom-up fashion.

Our Findings

f

Array vs Linked Structure for AVL

	Array-based	Linked Structure
Memory Usage	- Allocates a fixed-size array (or resizes dynamically) - May waste memory due to unused elements or resizing overhead	- Dynamic memory allocation only as needed - More efficient for sparse trees
Time Complexxity	- Insert: O(log n) - Delete: O(log n) - Search: O(log n) Harder to implement rotations	- Insert: O(log n) - Delete: O(log n) - Search: O(log n) Easy to implement recursive operations
Space Complexity	O(n), but may require extra space for reallocation or index shifting	O(n) (space grows with number of nodes)
Balance Factor Maintenance	- **More complex:** calculating child positions via indices requires more computation - Harder to perform rotations efficiently	- **Simple:** use pointers and maintain balance factor per node - Easy rotation logic using node pointers

Our Findings

g

Array vs LInked Structure for AVL

	Array-based	Linked Structure
Ease of Implementation	**Complex** Index management needed for parent/child references and rotations	**Simpler** Recursive structure naturally supports balancing logic
Cache Performance	**Better** Contiguous memory access improves cache locality	**Worse** Non-contiguous memory access due to dynamic allocation
Scalability	Limited by array resizing overhead or fixed size	Scales well with system memory
Summary	Slightly better cache performance but harder to manage and rotate.	Easier to maintain, scale, and rotate.Better suited for dynamic and recursive operations in trees.

REFERENCES

1. One Drive link

Group Members. (2025). Shared dataset folder for sorting and searching algorithm experiments [OneDrive folder]. Multimedia University. https://mmuedumy-my.sharepoint.com/:f:/g/personal/1211108003_student_mmu_edu_my/EmzGjVab8o9Fvr_OaeoC7JcBSTOWvZm1jzGu0aUhJWhrOQ?e=CgAS0l

2. Code Repository

Group Members. (2025). Source code for sorting and searching algorithm experiments [Google Colab notebook]. https://colab.research.google.com/drive/1DCidNNsHN76Hr8bSdJ4NSuNp4-oIO-DE?usp=sharing

3. Individual Runtime Documentation

- Tang, W. X. (2025). Individual experimental runtime report [Google Docs]. https://docs.google.com/document/d/1ZmtmvPggbOK6mpp-7bnD-qlny8RORWeB5Vt1gyGvZqY/edit?usp=sharing
- Tan, J. R. Y. (2025). Individual experimental runtime report [Google Docs]. https://docs.google.com/document/d/1czsY8Bse3y07fGQHPUP53VkMzyN-H0OrNLA0kBnRWGE/edit?tab=t.xysobmxqmplt
- Yeong, Z. Y. (2025). Individual experimental runtime report [Google Docs]. https://docs.google.com/document/d/1jAJ10TxiO1zrWS7Sh0GG5L7V6WV1bGelCnBnJdL6Z3Q/edit?usp=sharing
- Low, W. J. (2025). Individual experimental runtime report [Google Docs]. https://docs.google.com/document/d/1ld_h9-AAiOnJihaeB2y6XdE0i-h7UCnW_ByxEslJtzs/edit?usp=sharing

4. Group Excel File

Group Members. (2025). Group experimental study results [Microsoft Excel file]. Multimedia University SharePoint. https://mmuedumy-my.sharepoint.com/:x:/g/personal/1211112069_student_mmu_edu_my/EUA4ezlarQ9ItXf8v1HT_BwB4J2Nx7tTYeI-48B8NDzShQ?e=XAgSed

5. Individual Excel Files

- Tan, J. R. Y. (2025). Individual experimental data (Excel) [Microsoft Excel file]. Multimedia University SharePoint. https://mmuedumy-my.sharepoint.com/:x:/g/personal/1211108003_student_mmu_edu_my/Ef7T6sjcdrNIm5PYYkzGMKQBrDmdXAIrmog2Sau8oKqE5w?e=3pSiNd
- Yeong, Z. Y. (2025). Individual experimental data (Excel) [Microsoft Excel file]. Multimedia University SharePoint. https://mmuedumy-my.sharepoint.com/:x:/r/personal/1211107904_student_mmu_edu_my/Documents/Experiment_study_ade.xlsx?d=wb96d636b74964b2a8ce9d3ba942a6783&csf=1&web=1&e=LKfGkE
- Low, W. J. (2025). Individual experimental data (Excel) [Microsoft Excel file]. Multimedia University SharePoint. https://mmuedumy-my.sharepoint.com/:x:/r/personal/1211108404_student_mmu_edu_my/Documents/experimental%20study%20-%20wanjin.xlsx?d=w8dfe81deefb44d08afa2b1ff243003e2&csf=1&web=1&e=RKevPI

TT2L G9

1211112069 - Tang Wei Xiong

1211108003 - Joey Tan Rou Yi

1211107904 - Yeong Zi Yan

1211108404 - Low Wan Jin