ALGORITHM DESIGN AND ANALYSIS - ASSIGNMENT

GROUP D

Lecture Section: TC2L
Tutorial Section: TT5L
Trimester: March/April 2025 (Term 2510)

Name	ID
Muhammad Faiz bin Ilyasa	1221309435
Mohammed Yousef Mohammed Abdulkarem	1221305727
Mohammed Aamena Mohammed Abdulkarem	1221305728
Danish bin Ahmed Shahreeza	1211111869

Mr. Wong Keng Tuck

OVERVIEW

- Objective:
  - To Implement array-based AVL search using MergeSort, QuickSort, and Binary Search.
  - Done by sorting the array, then using binary-search to search for target in sorted array.

- Key Requirements:
  - Comparative analysis of Merge Sort VS. Quick Sort VS. Binary Search
  - Implement in 2 programming languages (Java + Python)
  - No sorting or searching library is used, and no internal sorting in a data structure (e.g. TreeSet or PriorityQueue)
  - Perform experimental study in different devices and record the data.

- Algorithms to be implemented:
  - **merge_sort_step** *(to list merge sorting steps in a .txt file)*
  - **quick_sort_step** *(to list quick sorting steps in a .txt file)*
  - **binary_search_step** *(to list search path for target in a .txt file)*
  - **dataset_generator** *(to generate n randomized unique elements in a .csv file)*
  - **merge_sort** *(to list all elements in a dataset in a sorted order and capturing the runtime)*
  - **quick_sort** *(to list all elements in a dataset in a sorted order and capturing the runtime)*
  - **binary_search** *(to list the running times for best, average, and worst case scenarios)*

DATASET GENERATOR

- We created a dataset_generator in java to generate *n* size of data in the format of integers and strings, separated by a comma.
- E.g: 1981761604, uoren
- The integers generated are unique, random, and up to at least 1 billion, while the strings are 5-6 randomized letters.
- The generator outputs in the form of : **dataset_*n*.csv**
- We generated 10 dataset files with the following sizes:

1000	1,000,000
10,000	10,000,000
50,000	50,000,000
100, 000	100,000,000
500,000	120,000,000

Sample dataset:

```
1   2139441538,ircaz
2   1528565345,lpqzf
3   1222019939,qflrv
4   862427528,nxeue
5   654341250,dmsfa
6   370691961,eyenv
7   1491535616,dtjxi
8   1466874316,dsovy
9   1234417850,qfzsf
10  658324376,jdqpl
```

MERGE-SORT ANALYSIS

Merge Sort is a classic divide-and-conquer sorting algorithm known for its stability and guaranteed performance. It operates by recursively dividing the dataset into halves, recursively sorting each half, and then efficiently merging the two sorted halves into a single, sorted list.

Algorithm Overview

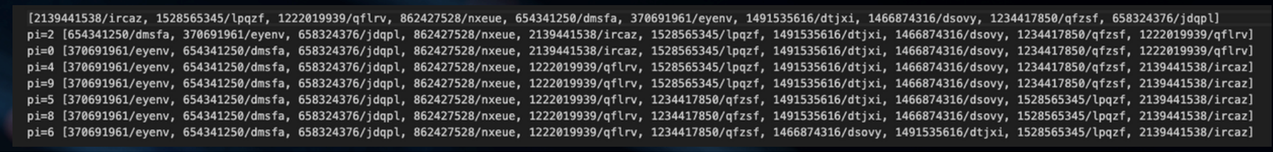
1. Divide the input array into two halves.
2. Recursively sort both halves.
3. Merge the two sorted halves by comparing elements.
4. Repeat until the entire array is sorted.

Implementation Details

- Language: Python and Java.
- Sorting Basis: Integer key within (integer, string) pairs.
- Outputs: Step-by-step merge process (merge_sort_step_start_end.txt), final sorted data (merge_sort_n.csv), and console runtime.

Case	Time Complexity	Explaination
Best	O(n log n)	Consistent efficient division and merging.
Average	O(n log n)	Consistent performance due to balanced splits.
Worst	O(n log n)	Guaranteed performance from full recursion and merging.
Space	O(n)	Requires auxiliary array for merging.

QUICK-SORT ANALYSIS



Quick Sort is an effective sorting algorithm that uses the divide and conquer concept and is based on comparisons. A pivot (the final element in this implementation) is chosen, the array is divided into sub-arrays containing elements smaller and larger than the pivot, and these sub-arrays are then sorted recursively.

Algorithm Overview

- Pivot Selection: Choose the last element as the pivot.
- Partitioning: Rearrange elements so smaller ones are left of the pivot, larger ones are right.
- Recursion: Apply steps above to left and right sub-arrays.
- Combine: Merge sorted partitions to finalize the array.

Implementation Details

- Language: Python and Java.
- Pivot: Last element.
- Sorting Basis: Integer key within (integer, string) pairs.
- Outputs: Step-by-step process (quick_sort_step_start_end.txt), final sorted data (quick_sort_n.csv), and console runtime.

Case	Time Complexity	Explaination
Best	O(n log n)	Evenly balanced partitions.
Average	O(n log n)	Typically balanced with random input.
Worst	O(n^2)	Unbalanced partitions (pivot is consistently min/max).
Space	O(log n)	Due to recursive call stack.

BINARY-SEARCH ANALYSIS

Binary Search works by dividing the interval in half. Of the value of search key is less than the middle interval, it will narrow the interval to lower half. Else it narrows to upper half.

Algorithm Overview

- Compare target to the middle
- If target is equal, return index
- If target is less, continue to the left
- If target is more, continue to the right
- Repeat until target is found or Empty

Implementation:

Language: Java and Python

Requirement: Input from Merge Sort

Output: binary_search_step<target>.txt

binary_search_n.txt

binary_search_step_613479842.txt

```
499: 1027377159/biaog
249: 511138138/zgeiv
374: 775715747/wiaph
311: 632734848/upiru
280: 571645541/edss
295: 598159169/fuhiz
303: 614628178/yoakm
299: 601405682/ncoago
301: 613151824/riqre
302: 613905556/jdaai
-1
```

outputs > binary_search_1000.txt

```
Best case total time: 0.362 ms
Average case total time: 0.646 ms
Worst case total time: 0.357 ms
```

Case	Time Complexity	Explaination
Best	O(1)	Target found on first comparison
Average	O(log n)	Halves the search space for each comparison
Worst	O(log n)	Either target if not present or at the end of array
Space	O(1)	No extra space used

EXPERIMENTAL SETUP

- The same dataset is used to compare every implemented algorithms, with size ranging from 1,000-120,000,000.
- However, different devices with different hardwares are used to get various execution times and performance results.
- Java and Python were chosen as the programming languages to carry out the experiment.
- Metrics:
  - Execution time (exclude I/O)
  - Steps logged (for _step versions)
- Link to datasets: https://drive.google.com/drive/folders/1M24vEZ_2vBktXDHHt73fv5M2QuSQkSth?usp=sharing

DEVICES SPECS

Device 1

Device name	Ezzie
Processor	AMD Ryzen 7 5700X 8-Core Processor 3.40 GHz
Installed RAM	16.0 GB
Device ID	C06E4317-8F60-449D-A1BA-7D41FAA02867
Product ID	00331-10000-00001-AA567
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

Device 2

Device name	ImYousef1l
Processor	13th Gen Intel(R) Core(TM) i7-13700HX 2.10 GHz
Installed RAM	16.0 GB (13.7 GB usable)
Device ID	8E5897A2-935A-44F2-92D9-4E8DEC5B45D2
Product ID	00342-42639-81703-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

Device 3

MacBook Pro

14-inch, 2021

Chip	Apple M1 Pro
Memory	16 GB
Serial number	GGW9M3P790
macOS	Sequoia 15.5

More Info...

Regulatory Certification

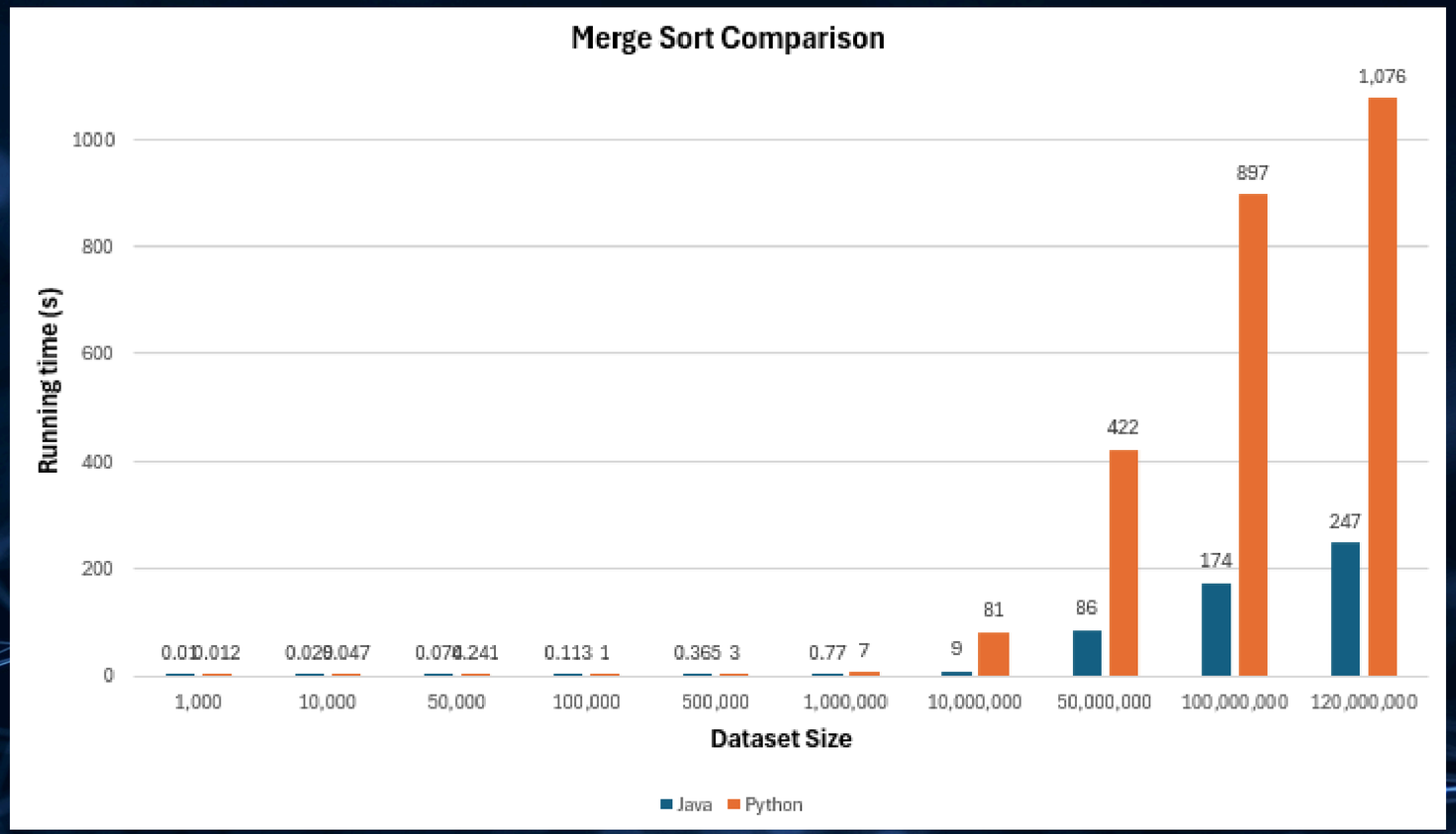
Device 4

Device name	LAPTOP-D9AOGL80
Processor	12th Gen Intel(R) Core(TM) i5-1235U 1.30 GHz
Installed RAM	8.00 GB (7.68 GB usable)
Device ID	50458D40-53BB-4F58-AE30-1989DA6D2579
Product ID	00356-24591-77991-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

DEVICE 1

MERGE-SORT

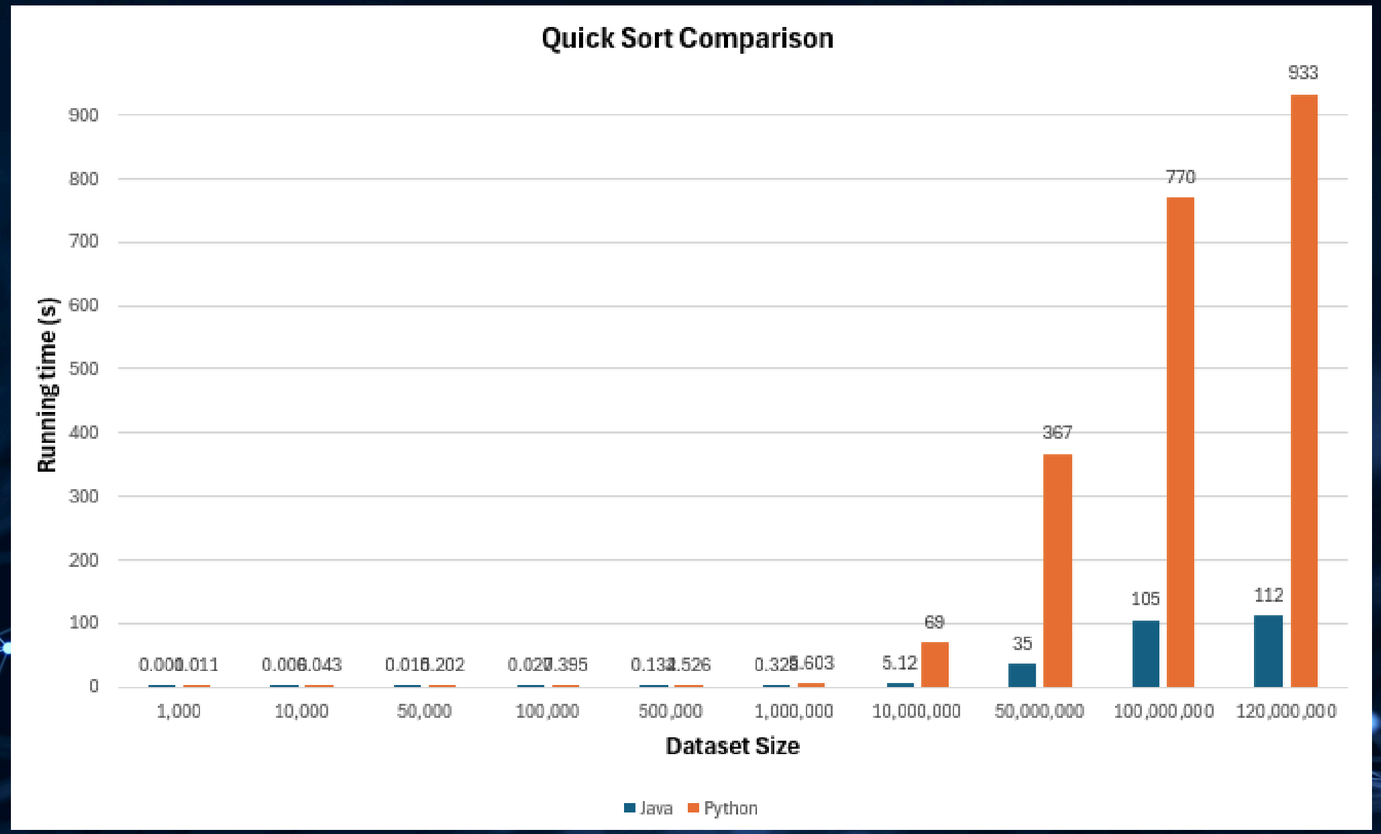
	Running time (ms)	
Dataset Size	Java	Python
1000	10	12
10000	29	47
50000	74	241
100000	113	531
500000	365	3162
1000000	770	6524
10000000	8973	80570
50000000	85993	421646
100000000	173840	897223
120000000	247387	1075584



DEVICE 1

QUICK-SORT

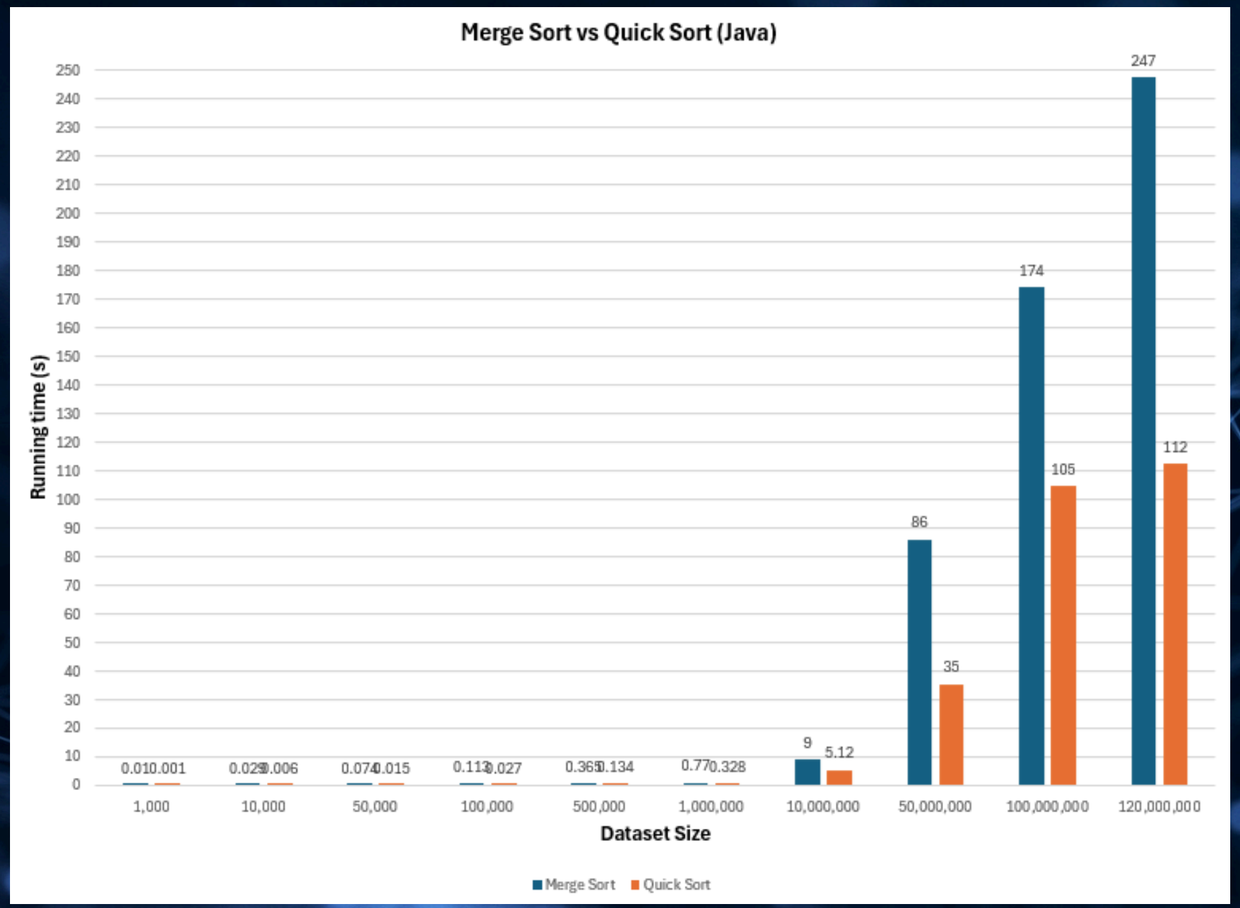
	Running time (ms)	
Dataset Size	Java	Python
1000	1	11
10000	6	43
50000	15	202
100000	27	395
500000	134	2526
1000000	328	5603
10000000	5120	69156
50000000	35184	366702
100000000	104541	769655
120000000	112245	932898



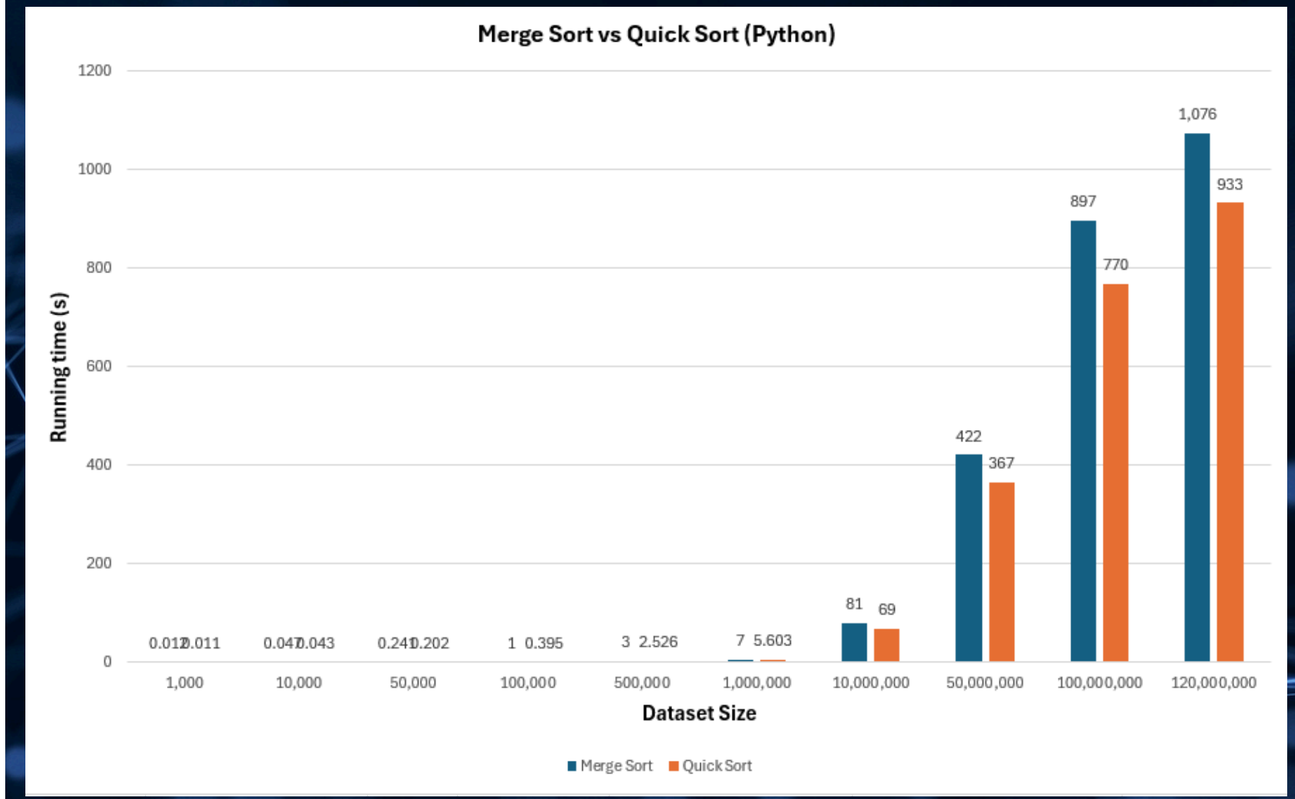
DEVICE 1

MERGE SORT VS QUICK SORT

JAVA



PYTHON



DEVICE 1

BINARY-SEARCH

JAVA

	Running time (ms)		
Dataset Size	Best	Average	Worst
1000	0.211	0.288	0.074
10000	1.215	1.597	0.784
50000	2.231	4.696	1.754
100000	2.573	7.646	3.08
500000	9.706	47.17	11.849
1000000	23.065	105.121	21.843
10000000	212.089	2277.681	250.622
50000000	1169.507	22087.711	1380.834
100000000	2463.661	51330.871	2841.963
120000000	3010.055	63032.711	3408.768

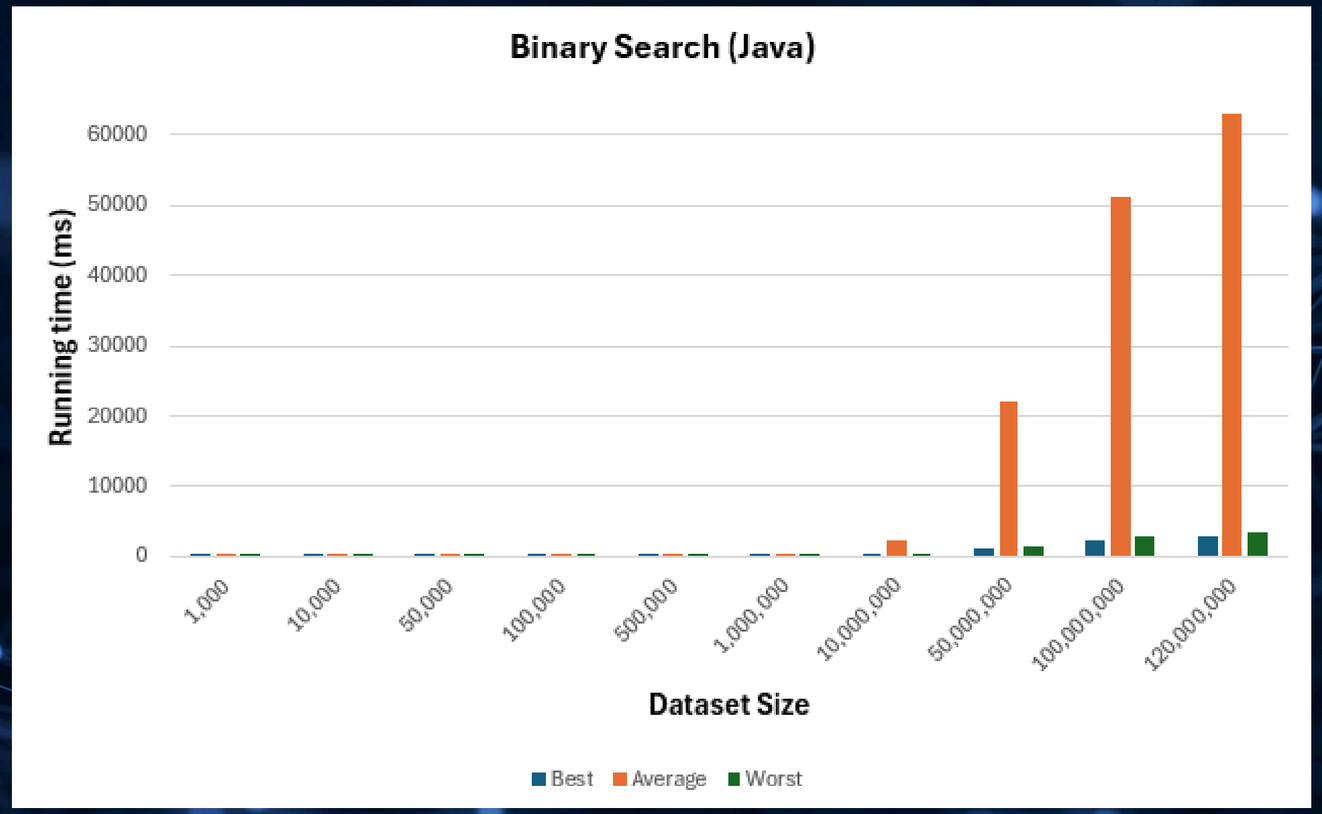
PYTHON

	Running time (ms)		
Dataset Size	Best	Average	Worst
1000	0.987	1.653	1.262
10000	12.445	16.315	15.404
50000	70.082	93.814	89.907
100000	152.121	203.084	187.669
500000	843.317	1247.321	1026.221
1000000	1815.461	2795.108	2181.461
10000000	21692.613	35574.983	26461.772
50000000	102838.748	177514.032	127784.781
100000000	193959.533	354727.873	252428.964
120000000	293894.321	660118.267	349568.663

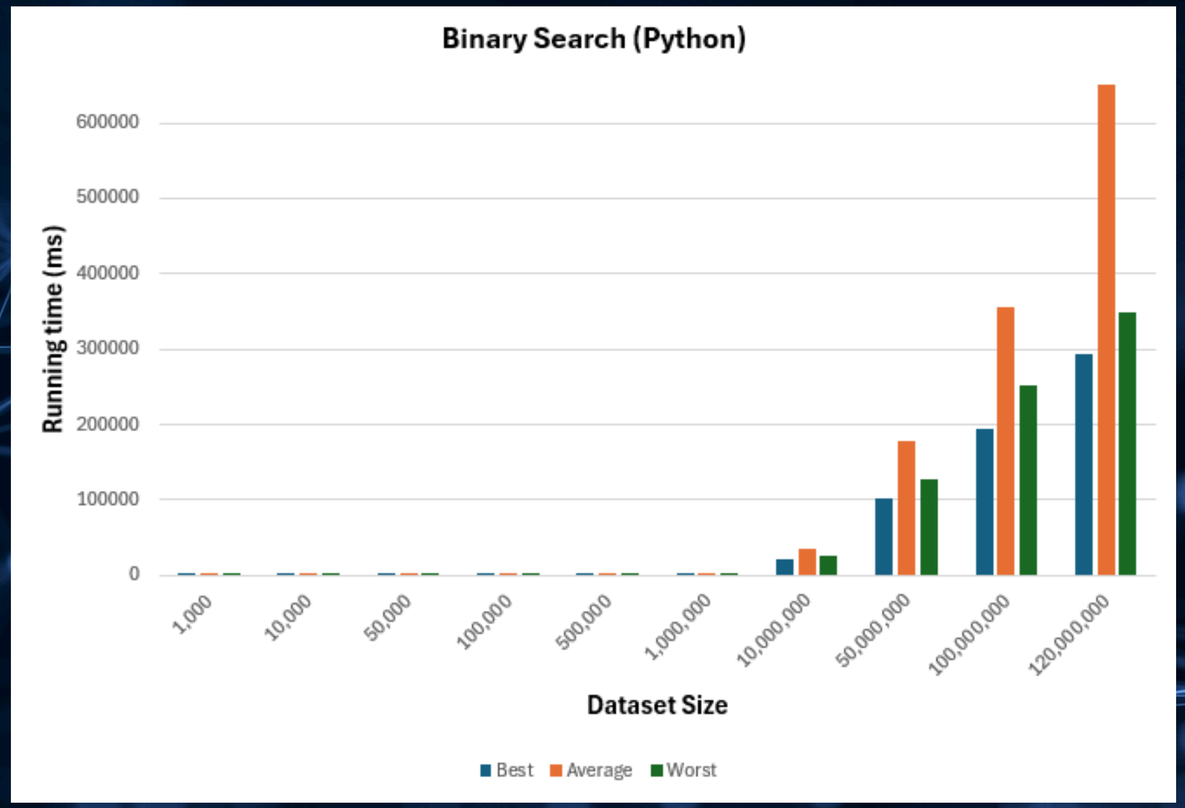
DEVICE 1

BINARY-SEARCH

JAVA



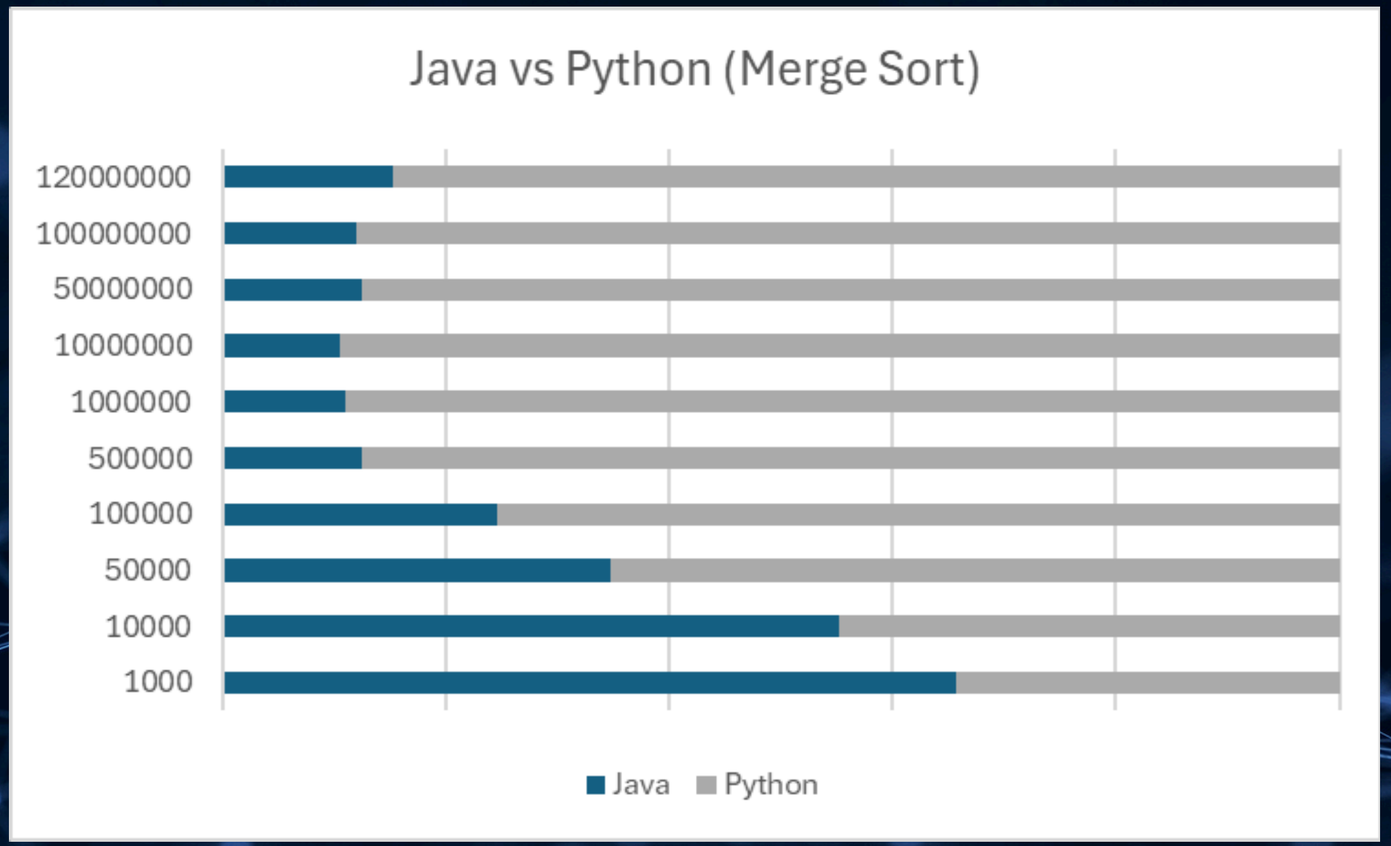
PYTHON



DEVICE 2

MERGE-SORT

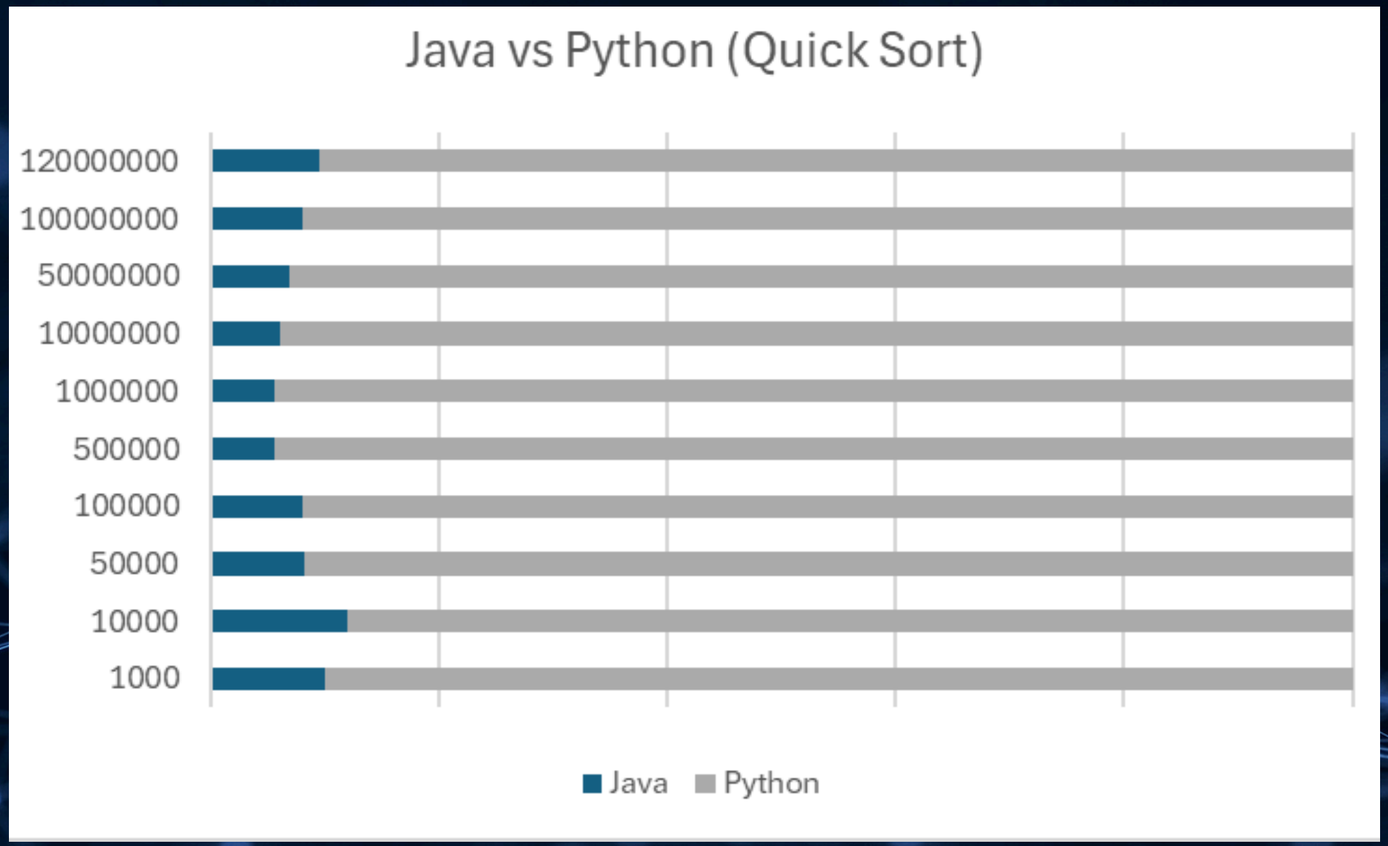
	Running time (ms)	
Dataset Size	Java	Python
1000	23	12
10000	53	43
50000	107	201
100000	134	411
500000	391	2758
1000000	697	5682
10000000	7910	67938
50000000	52051	366479
100000000	114318	845530
120000000	184346	1026023



DEVICE 2

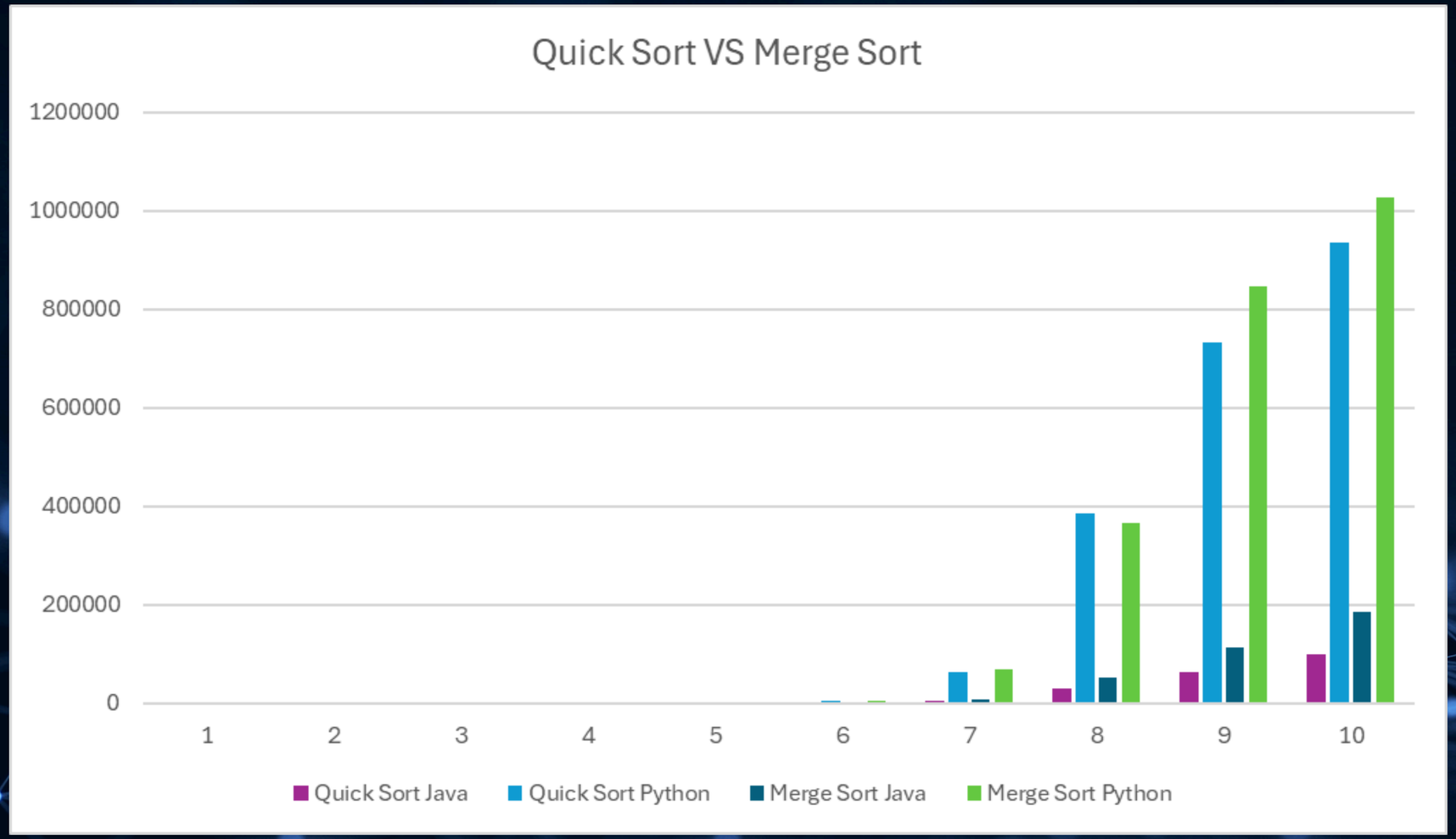
QUICK-SORT

	Running time (ms)	
Dataset Size	Java	Python
1000	1	9
10000	5	37
50000	15	166
100000	29	332
500000	128	2183
1000000	286	4894
10000000	4089	63529
50000000	28504	384498
100000000	64101	733954
120000000	98874	935748



DEVICE 2

MERGE SORT VS QUICK SORT



DEVICE 2

BINARY-SEARCH - MERGE SORT

JAVA

	Running time (ms)		
Dataset Size	Best	Average	Worst
1000	0.214	0.308	0.063
10000	1.006	1.723	0.773
50000	3.936	5.014	4.748
100000	7.643	9.879	2.868
500000	9.904	49.662	9.921
1000000	17.945	113.695	19.533
10000000	205.751	1722.284	224.208
50000000	1010.470	8811.839	1137.581
100000000	1985.947	17075.430	2226.666
120000000	2435.790	20942.293	3312.950

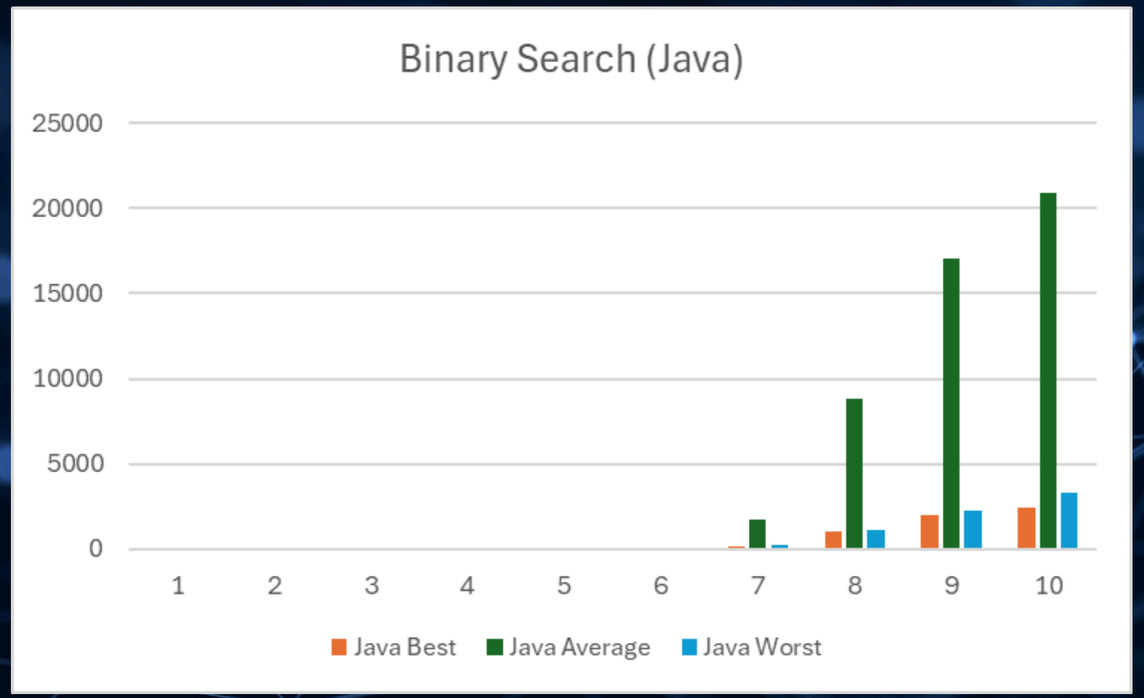
PYTHON

	Running time (ms)		
Dataset Size	Best	Average	Worst
1000	0.657	0.927	0.835
10000	8.955	11.958	11.328
50000	51.933	72.601	79.145
100000	110.341	150.042	133.705
500000	613.028	914.629	759.114
1000000	1320.189	2055.391	1633.471
10000000	16755.416	30098.124	20328.556
50000000	89114.887	232650.104	108371.559
100000000	358844.225	478741.977	241065.488
120000000	231683.241	605042.311	275016.432

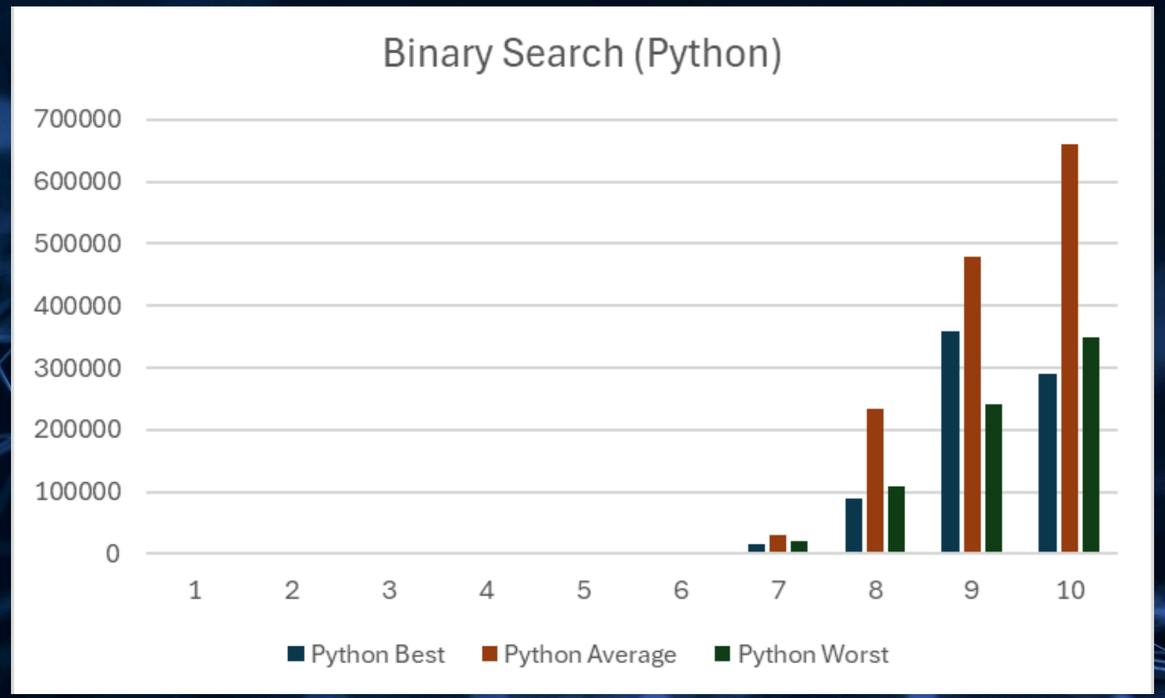
DEVICE 2

BINARY-SEARCH - MERGE SORT

JAVA



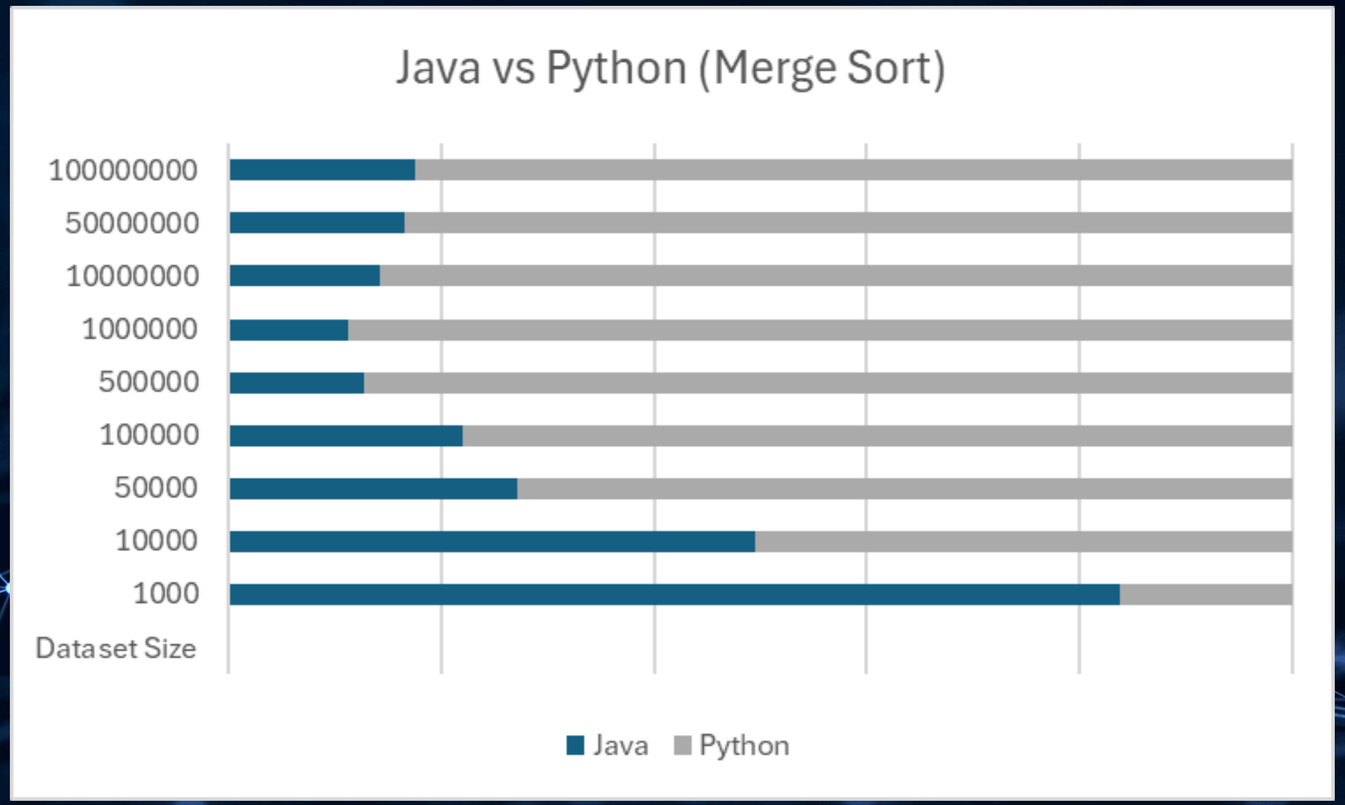
PYTHON



DEVICE 3

MERGE-SORT

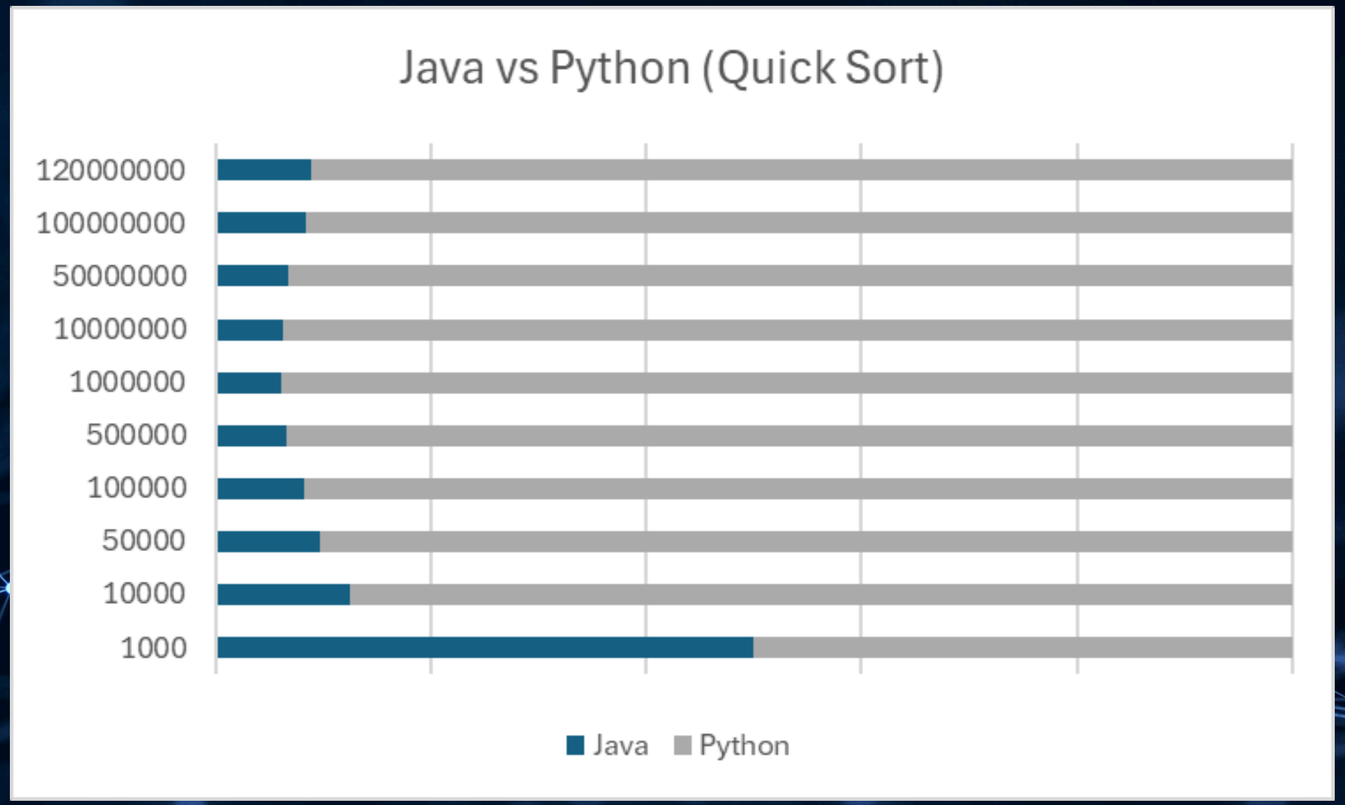
	Running time (ms)	
Dataset Size	Java	Python
1000	26	5
10000	46	47
50000	92	247
100000	144	511
500000	423	2879
1000000	793	6232
10000000	12061	72698
50000000	78967	399839
100000000	186859	879645
120000000	243603	1055574



DEVICE 3

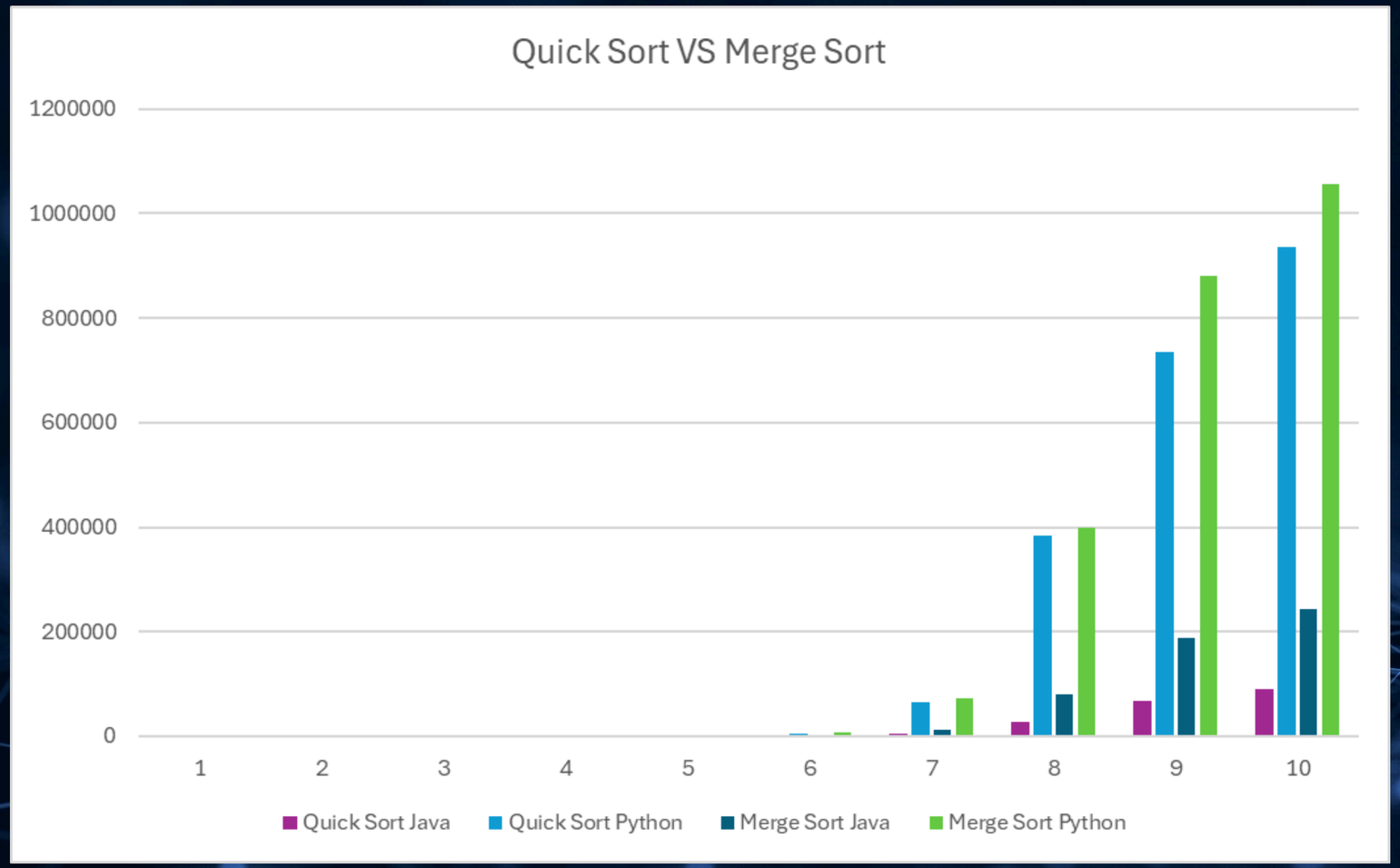
QUICK-SORT

	Running time (ms)	
Dataset Size	Java	Python
1000	4	4
10000	5	35
50000	18	168
100000	31	346
500000	148	2134
1000000	309	4787
10000000	4240	58871
50000000	27480	317621
100000000	67332	640554
120000000	91369	770298



DEVICE 3

MERGE SORT VS QUICK SORT



DEVICE 3

BINARY-SEARCH - QUICK SORT

JAVA

	Running time (ms)		
Dataset Size	Best	Average	Worst
1000	0.337	0.711	0.09
10000	0.768	1.627	0.997
50000	4.012	6.485	2.417
100000	3.037	13.479	4.023
500000	14.198	73.145	15.769
1000000	28.976	158.058	31.86
10000000	355.76	3081.135	384.878
50000000	1996.083	28151.647	2154.468
100000000	4198.633	67897.068	4515.346
120000000	5034.398	84767.521	5418.555

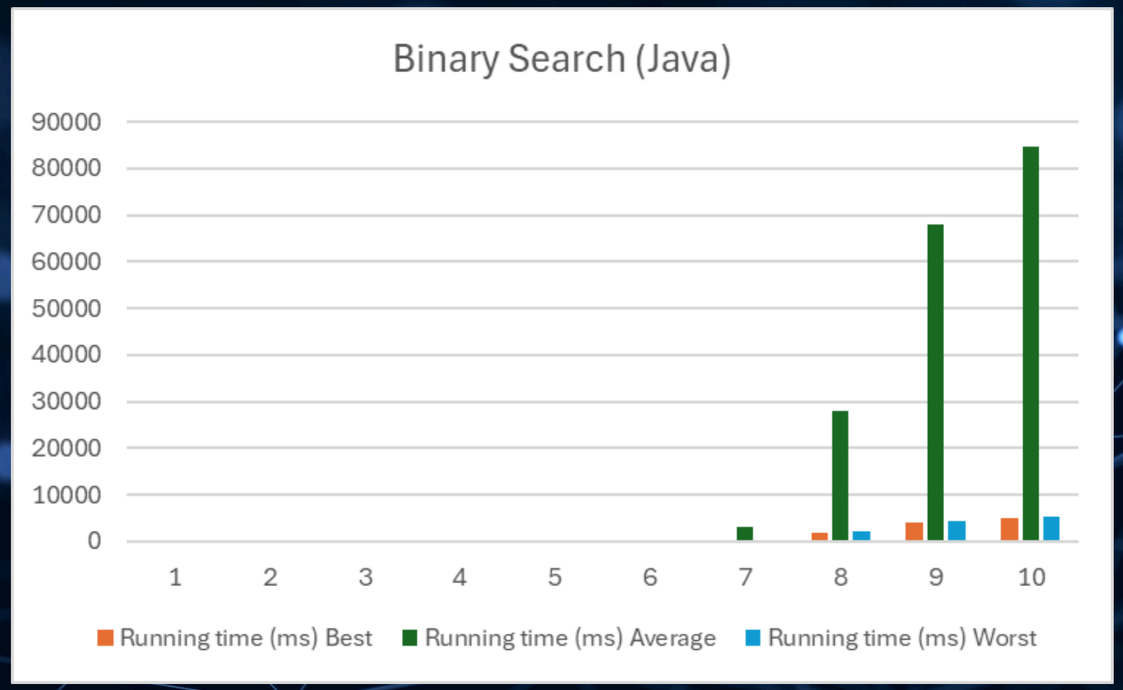
PYTHON

	Running time (ms)		
Dataset Size	Best	Average	Worst
1000	2.71	3.583	2.384
10000	23.103	23.099	19.85
50000	95.286	124.755	111.688
100000	201.851	262.508	236.388
500000	1130.889	1514.952	1314.28
1000000	2390.684	3731.655	2770.895
10000000	28611.21	44750.864	32876.295
50000000	156653.809	281720.745	177757.925
100000000	313307.618	563441.49	355515.85
120000000	375969.142	676129.788	426618.999

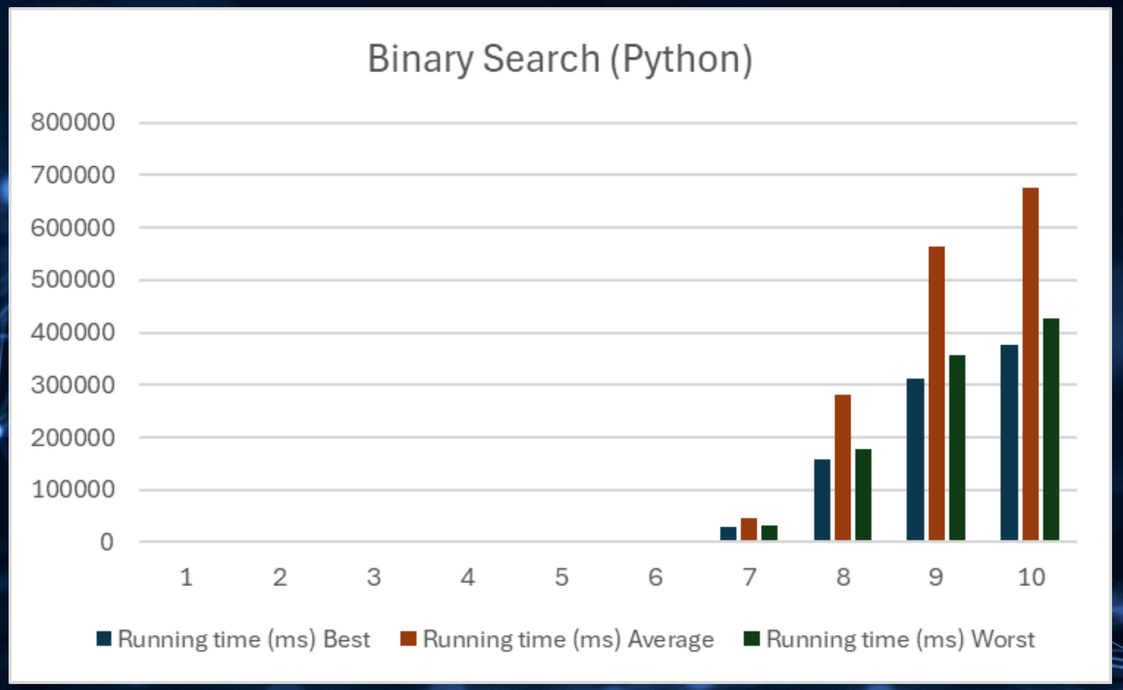
DEVICE 3

BINARY-SEARCH

JAVA



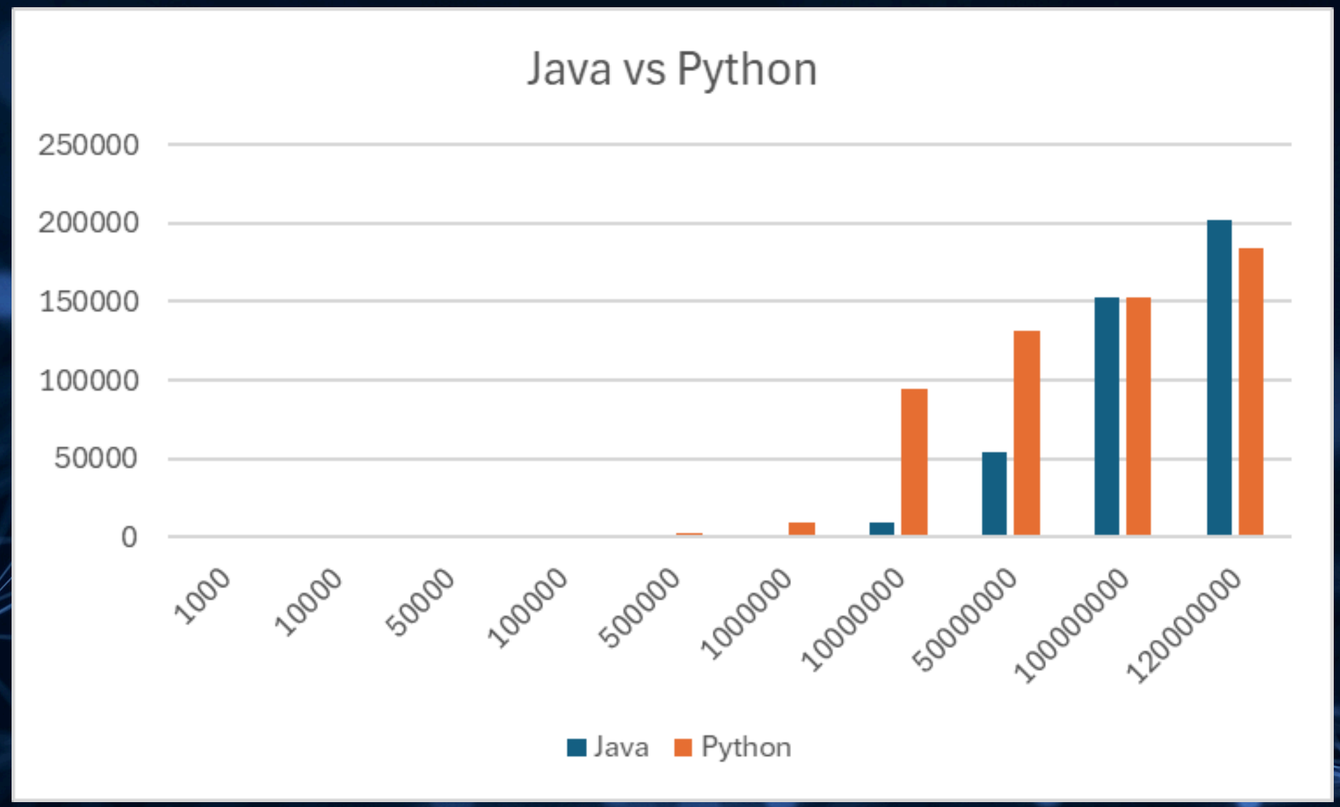
PYTHON



DEVICE 4

MERGE-SORT

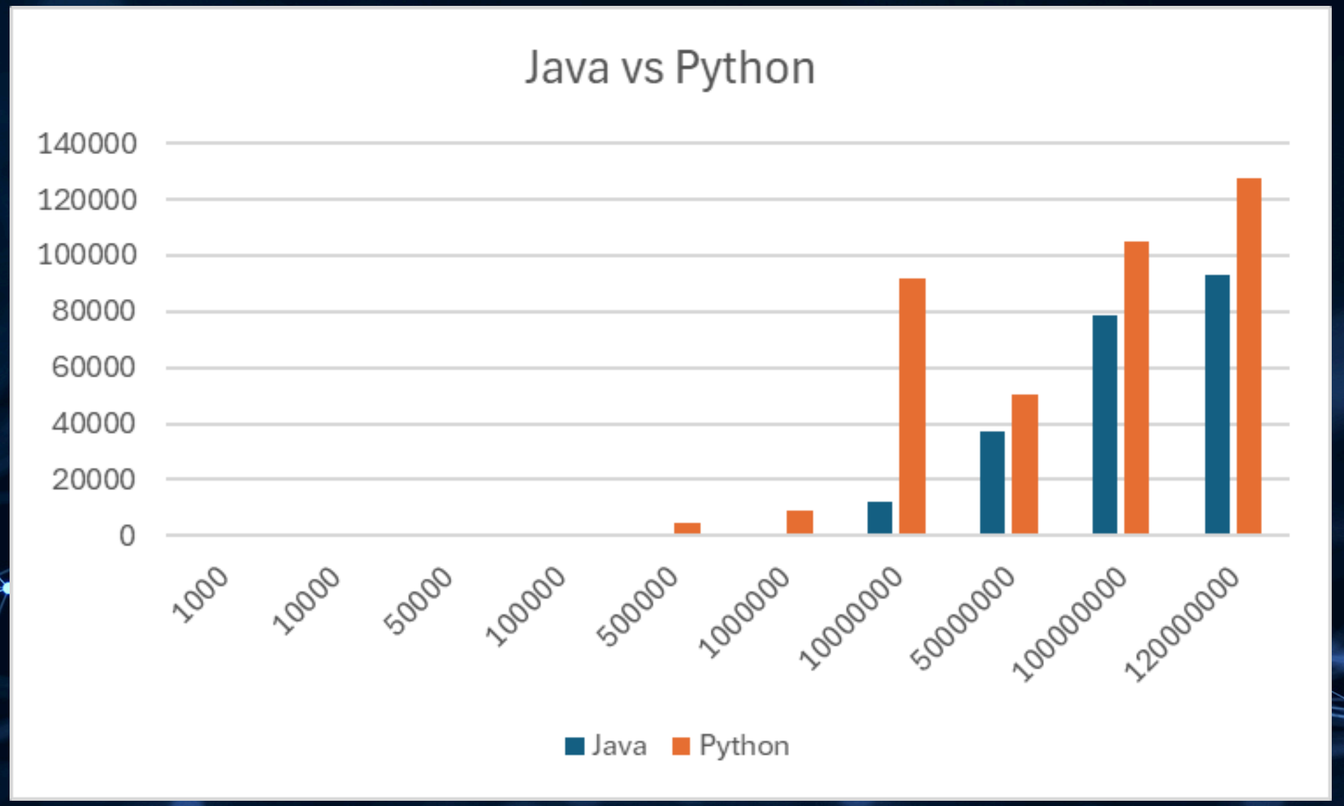
	Running time (ms)	
Dataset Size	Java	Python
1000	54	44
10000	135	71
50000	190	333
100000	289	391
500000	806	2466
1000000	1459	9151
10000000	8808	94677
50000000	53879	131709
100000000	152850	152233
120000000	201734	184377



DEVICE 4

QUICK-SORT

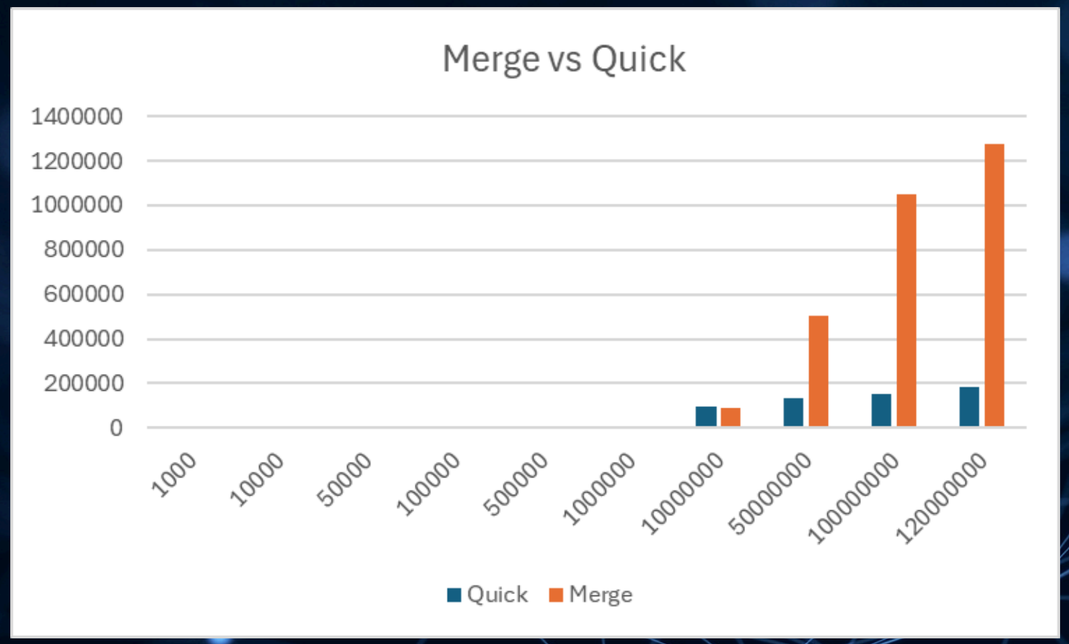
	Running time (ms)	
Dataset Size	Java	Python
1000	26	21
10000	43	62
50000	74	231
100000	94	493
500000	261	4844
1000000	734	9058
10000000	12099	91845
50000000	37396	505148
100000000	78594	105051
120000000	93284	127266



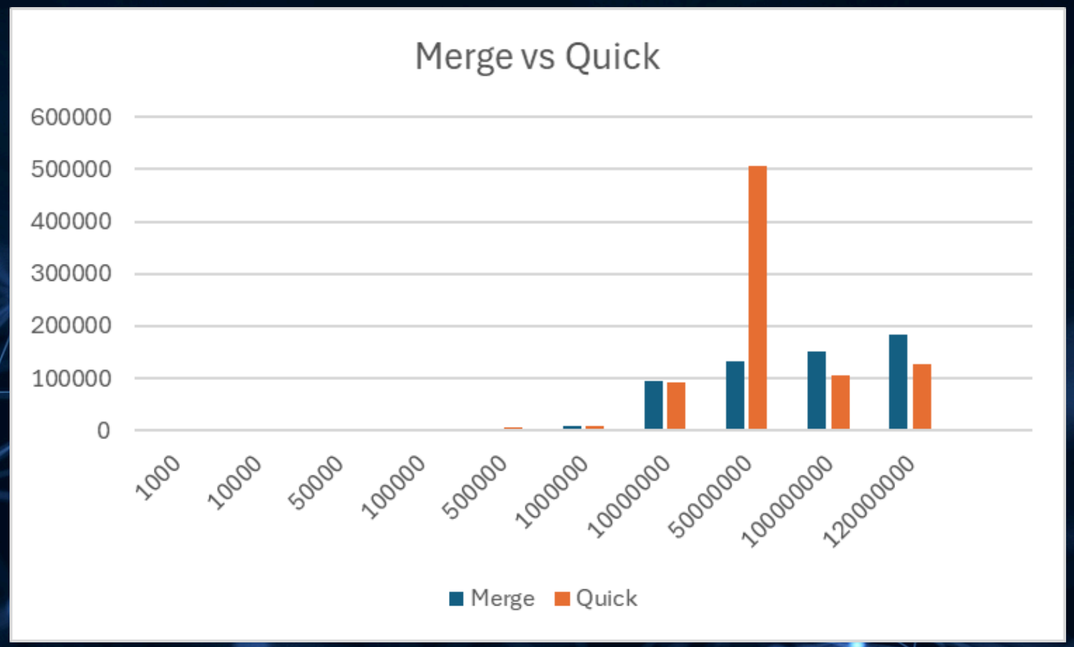
DEVICE 4

MERGE SORT VS QUICK SORT

JAVA



PYTHON



DEVICE 4

BINARY-SEARCH

JAVA

	Running time (ms)		
Dataset Size	Best	Average	Worst
1000	0.342	0.476	0.311
10000	1.501	2.672	0.617
50000	4.307	7.974	1.421
100000	4.833	13.621	2.996
500000	15.613	46.174	15.481
1000000	34.618	143.026	27.653
10000000	304.02	1273.625	283.347
50000000	456.671	1971.171	475.612
100000000	3960.91	13032.361	2979.604
120000000	5766.308	20052.507	4860.936

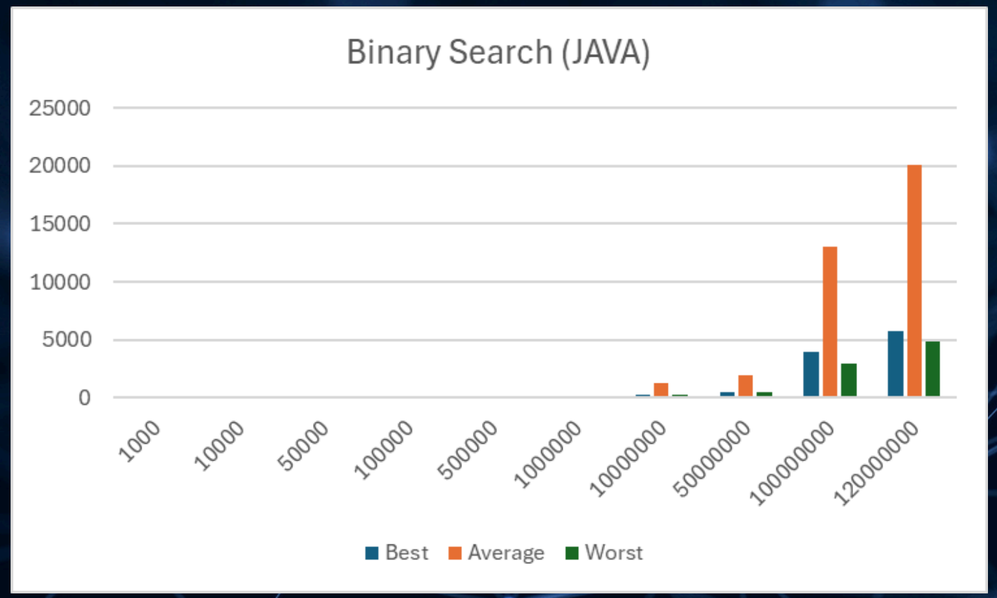
PYTHON

	Running time (ms)		
Dataset Size	Best	Average	Worst
1000	0.881	1.464	2.655
10000	10.014	14.732	13.407
50000	76.66	101.548	87.781
100000	134.802	253.96	230.629
500000	735.156	1563.866	962.396
1000000	1546.294	2861.28	2012.339
10000000	20836.007	41895.825	29287.167
50000000	50268.809	71926.847	36901.121
100000000	100537.618	143853.694	73802.242
120000000	120645.142	172624.433	88562.69

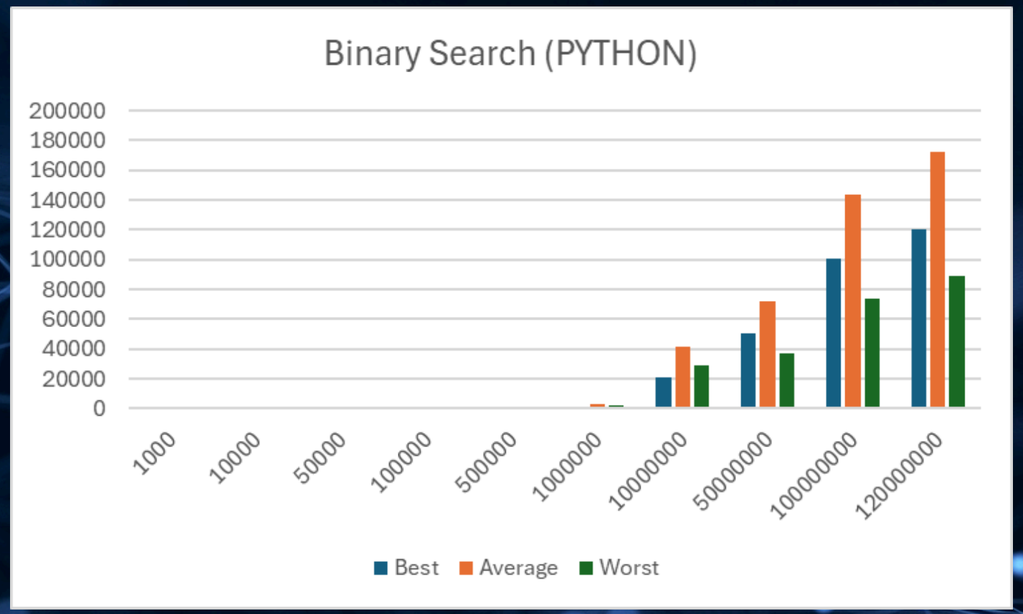
DEVICE 4

BINARY-SEARCH

JAVA



PYTHON



EXTRA FILES

- Mohammed Yousef Mohammed Abdulkarem - Running Samples - Device 2
- Mohammed Aamena Mohammed Abdulkarem - Running Samples - Device 3

CONCLUSION

- *By manually applying sorting algorithms, Quick Sort and Merge Sort in particular, to sections with substantial datasets, this assignment investigated them. According to the performance tests, Quick Sort continuously performed better than Merge Sort, particularly when manual merging and an in-place iterative approach were used. Merge Sort tends to have larger overhead even when it is stable. Additionally, we tested these methods across a range of hardware configurations and programming languages, including Python and Java. This shown that performance can be strongly impacted by the language's efficiency and memory management. In general, Python stands out for its simplicity and versatility, while Java has more object overhead but better memory control.*
- *Merge Sort is better suited for array-based AVL tree implementations due to its stable O(n log n) performance, which aligns well with sorted data operations. Quick Sort, while often faster, can introduce instability with skewed inputs.*
- *Array-based AVL trees offer faster indexed access and better cache performance but have higher costs for resizing and updates. In contrast, linked AVL trees support dynamic memory and efficient insert/delete operations but suffer from slower traversal and pointer overhead.*
- *Overall, this work highlights the trade-offs between data structures and sorting strategies, deepening understanding of scalable, memory-efficient design.*

REFERENCES

- GeeksforGeeks. (2025, May 12). Binary Search Algorithm iterative and Recursive Implementation. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/binary-search/
- GeeksforGeeks. (2025a, April 17). Quick sort. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/
- W3Schools.com. (n.d.). https://www.w3schools.com/dsa/dsa_algo_quicksort.php
- W3Schools.com. (n.d.-b). https://www.w3schools.com/dsa/dsa_algo_binarysearch.php
- W3Schools.com. (n.d.-c). https://www.w3schools.com/dsa/dsa_algo_mergesort.php
- GeeksforGeeks. (2025b, April 25). Merge Sort Data Structure and Algorithms tutorials. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/merge-sort/

CONTRIBUTIONS

- **merge_sort_step.java -** Mohammed Yousef Mohammed Abdulkarem
- **quick_sort_step.java -** Mohammed Aamena Mohammed Abdulkarem
- **binary_search_step.java -** Danish bin Ahmed Shahreeza
- **dataset_generator.java -** Muhammad Faiz bin Ilyasa
  - Mohammed Yousef Mohammed Abdulkarem
  - Mohammed Aamena Mohammed Abdulkarem
  - Danish bin Ahmed Shahreeza
- **merge_sort.java -** Mohammed Yousef Mohammed Abdulkarem
- **quick_sort.java -** Mohammed Aamena Mohammed Abdulkarem
- **binary_search.java -** Muhammad Faiz bin Ilyasa

- **merge_sort_step.py -** Muhammad Faiz bin Ilyasa
- **quick_sort_step.py -** Danish bin Ahmed Shahreeza
- **binary_search_step.py -** Danish bin Ahmed Shahreeza
- **merge_sort.py -** Mohammed Yousef Mohammed Abdulkarem
- **quick_sort.py -** Mohammed Aamena Mohammed Abdulkarem
- **binary_search.py -** Muhammad Faiz bin Ilyasa


THANK YOU
QUESTIONS?