PROJECT ASSIGNMENT

Group 7| T21L | Ms Nur Erlida Binti Ruslan

Algorithm Design & Analysis

Name	Student ID	Contribution
1. AZZRUDDIN BIN AZZUDDIN	1211112098	25%
2. MOHAMAD AMIRUL HAZIEQ BIN MOHAMAD ZAINI	1211112352	25%
3. MUHAMMAD AMEERUL ARIEFF HAKIMIE BIN ABDUL SUKOR	1231303379	25%
4. MUHAMMAD IQBAL HAKIM BIN ISMAIL (LEADER)	1211112101	25%

Introduction

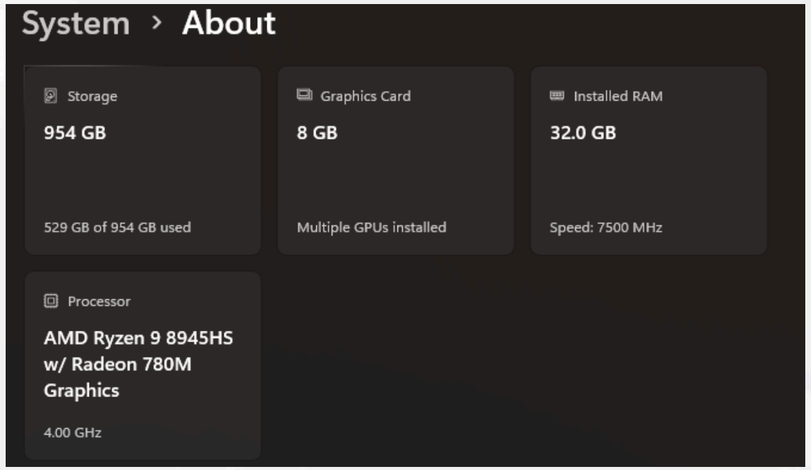
In this project, we focus on three important algorithms: Merge Sort, Quick Sort, and Binary Search. These are widely used for organizing and searching data efficiently.

We implement the algorithms in Python and Java, compare their performance, and analyze their time and space complexity. Binary Search is also used to simulate searching in an AVL Tree.

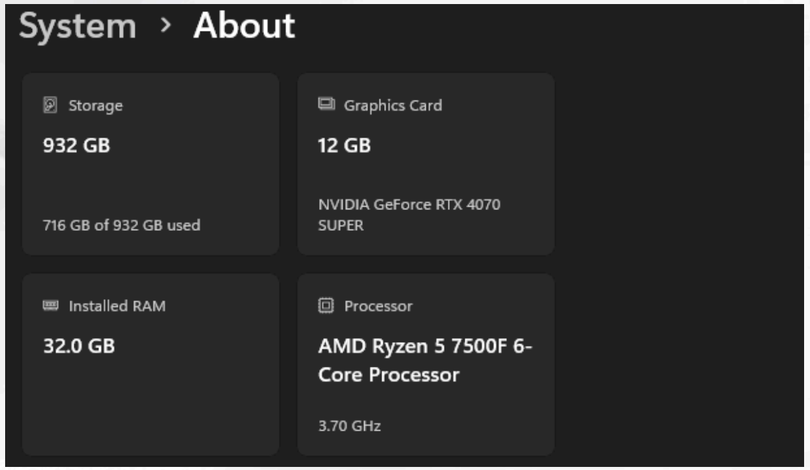
The idea is to see how these algorithms do, how fast are them and how do they perform in relation to the data size. This allows us to exert a strong emphasis on the design of professional algorithms and problemsolving.

DEVICE SPECIFICATIONS

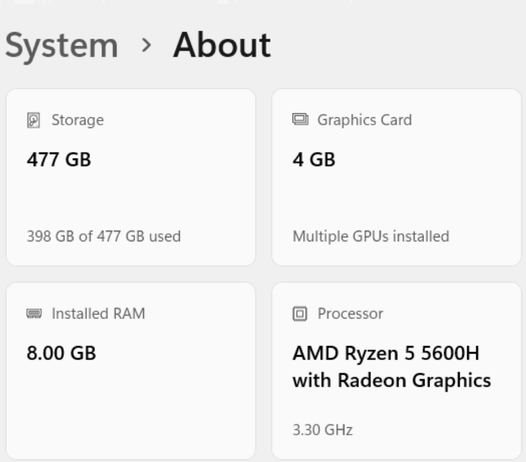
IQBAL



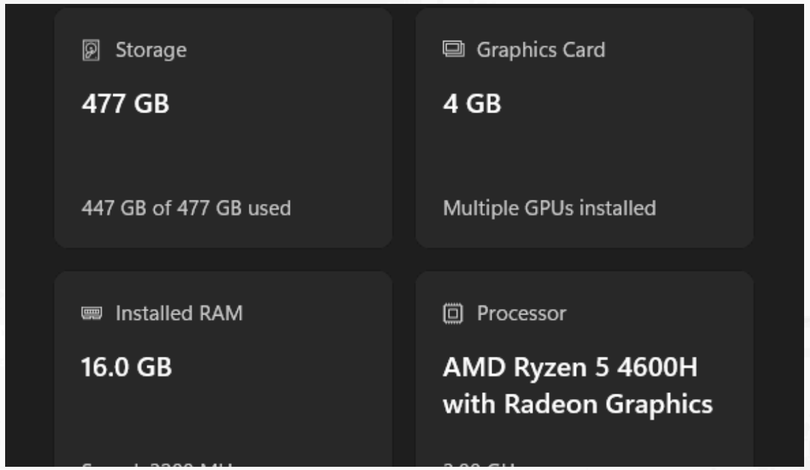
AMEERUL



AMIRUL



AZZRUDDIN



Links to Dataset (Onedrive of Each Person)

NAME	DATASET	ONEDRIVE LINKS
AZZRUDDIN	1000000,3000000,5000000, 9000000,11000000,13000000,15000000, 17000000,19000000,45000000	https://mmuedumy-my.sharepoint.com/:f:/g/personal/1211112098_student_mmu_edu_my/EgxMxYbmnclMqrJRV6kz19wBNoKeh_H9ABx-npLEK7TMEw?e=vWwj8T
AMIRUL HAZIEQ	3000000,5000000,7000000, 9000000,11000000,13000000,15000000, 17000000,19000000,50000000	https://mmuedumy-my.sharepoint.com/:f:/g/personal/1211112352_student_mmu_edu_my/EmAJH6KlK6hKsq2a7f7NPFABGtmFXPHFoBEx-le2y1phUg?e=iG8cw7
AMEERUL ARIEFF	2000000,3000000,4000000,5000000,6000000,7000000,8000000,9000000,10000000,100000000	https://mmuedumy-my.sharepoint.com/:f:/g/personal/1231303379_student_mmu_edu_my/EiM306KevbdEuu3apTt2_dcBR-no-7Ua1G_vMXr8urK7jw?e=Badgl6
IQBAL HAKIM	10000000,20000000,30000000,4000000, 50000000,60000000,70000000,80000000, 90000000,100000000	https://1drv.ms/f/c/810e1691468c2a02/EmLz2WHRPABMvKeL9kFD_G8Byv8yXca-Wn_YXZQq6zJg3Q?e=3NEPlZ

THEROTICAL ANALYSIS

MERGE SORT

Merge sort is **divide and conquer algorithm** that splits the array, sorts each half, and merges them back together

Time and Space Complexity :

- **Time :** O(n log n) for all cases
- **Space :** O(n) uses extra memory for merging

Pros :

1. Always O(n log n), even in worst case
2. Stable sort (keeps equal items in order)
3. Good for large datasets

Cons :

1. Uses extra memory (O(n))
2. Can be slower than Quick Sort on small arrays

CASE	DESCRIPTION	COMPLEXITY
Best Case	Data is already sorted	O(n log n)
Average Case	Randomly ordered data	O(n log n)
Worst Case	Data is in reverse order	O(n log n)

QUICK SORT

A **divide and conquer algorithm** that picks a **pivot**, partitions the array into **two parts** (less than and greater than the pivot), and recursively sorts them.

Time and Space Complexity :

- Time: Best/Average: O(n log n), Worst: $O(n^2)$
- Space: O(log n) for recursive calls (in-place)

Pros:

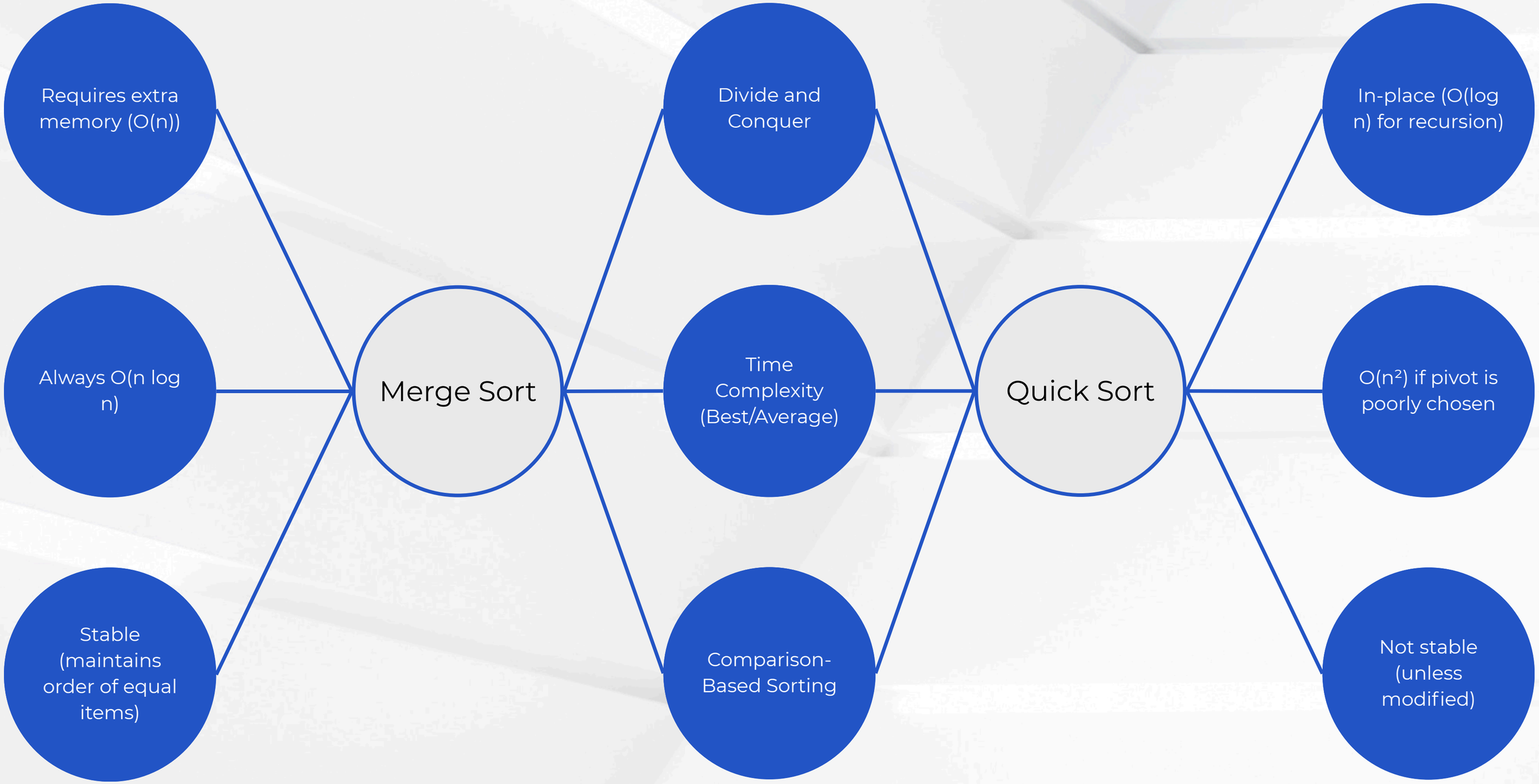
1. Faster than Merge Sort for small arrays
2. In-place (no extra space needed)
3. Efficient average-case performance

Cons

1. Worst-case is $O(n^2)$ if pivot choice is poor
2. Not stable
3. Requires good pivot strategy

CASE	DESCRIPTION	COMPLEXITY
Best Case	Pivot splits array into equal halves	O(n log n)
Average Case	Random pivot with balanced partitions	O(n log n)
Worst Case	Pivot is always smallest or largest element (unbalanced)	$O(n^2)$

THEORETICAL ANALYSIS



BINARY SEARCH

An efficient algorithm to find an item in a sorted array by **repeatedly** dividing the search interval in half.

Time and Space Complexity :

- Time: O(log n)
- Space: O(1) (iterative) or O(log n) (recursive)

Pros:

1. Very fast for large sorted data
2. Simple and efficient
3. Uses minimal comparisons

Cons :

1. Works only on sorted data
2. More complex to implement than linear search
3. Inefficient for small arrays or unsorted data

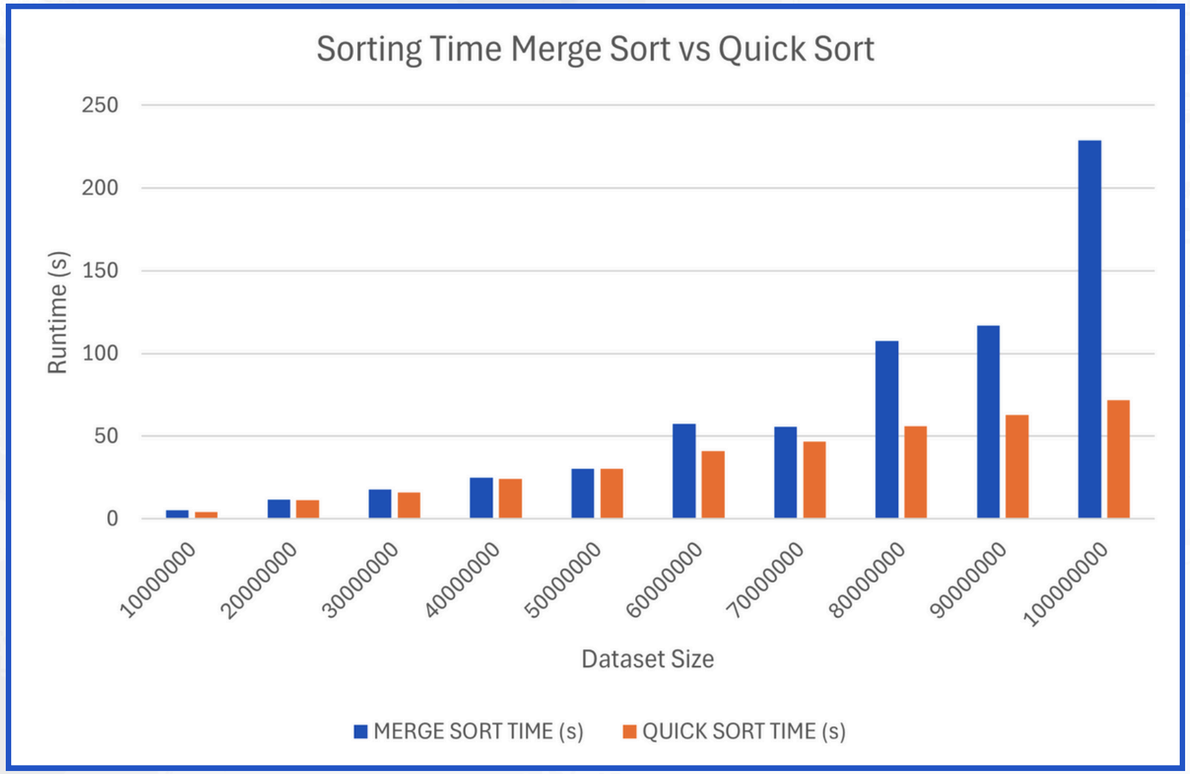
CASE	DESCRIPTION	COMPLEXITY
Best Case	Middle element is the target	O(1)
Average Case	Target found after a few splits	O(log n)
Worst Case	Target not found, then requires full depth search	O(log n)

EXPERIMENTAL ANALYSIS RESULT

SORTING RUNTIME COMPARISON (JAVA)

IQBAL

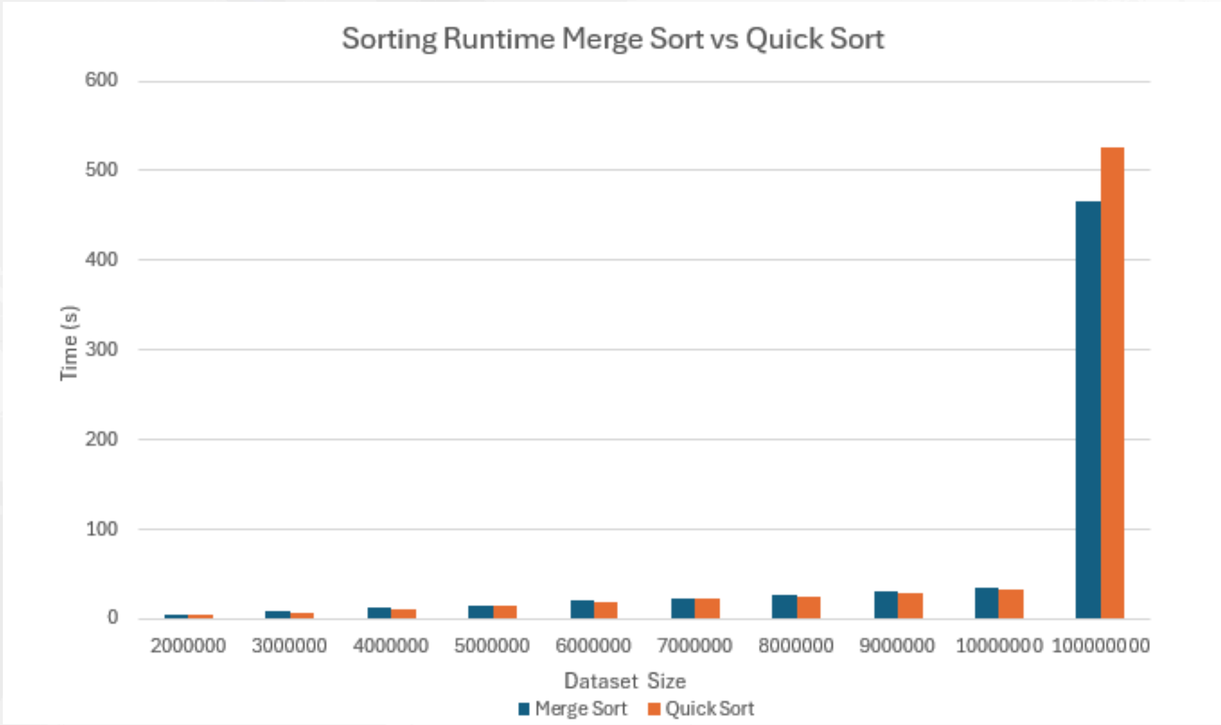
NO	DATASET	MERGE SORT TIME (s)	QUICK SORT TIME (s)
1	10000000	5.349	4.191
2	20000000	11.487	11.15
3	30000000	17.728	15.936
4	40000000	24.966	24.257
5	50000000	30.142	30.138
6	60000000	57.588	40.945
7	70000000	55.571	46.828
8	80000000	107.605	56.164
9	90000000	116.747	62.851
10	100000000	228.823	71.875



SORTING RUNTIME COMPARISON (PYTHON)

AMEERUL

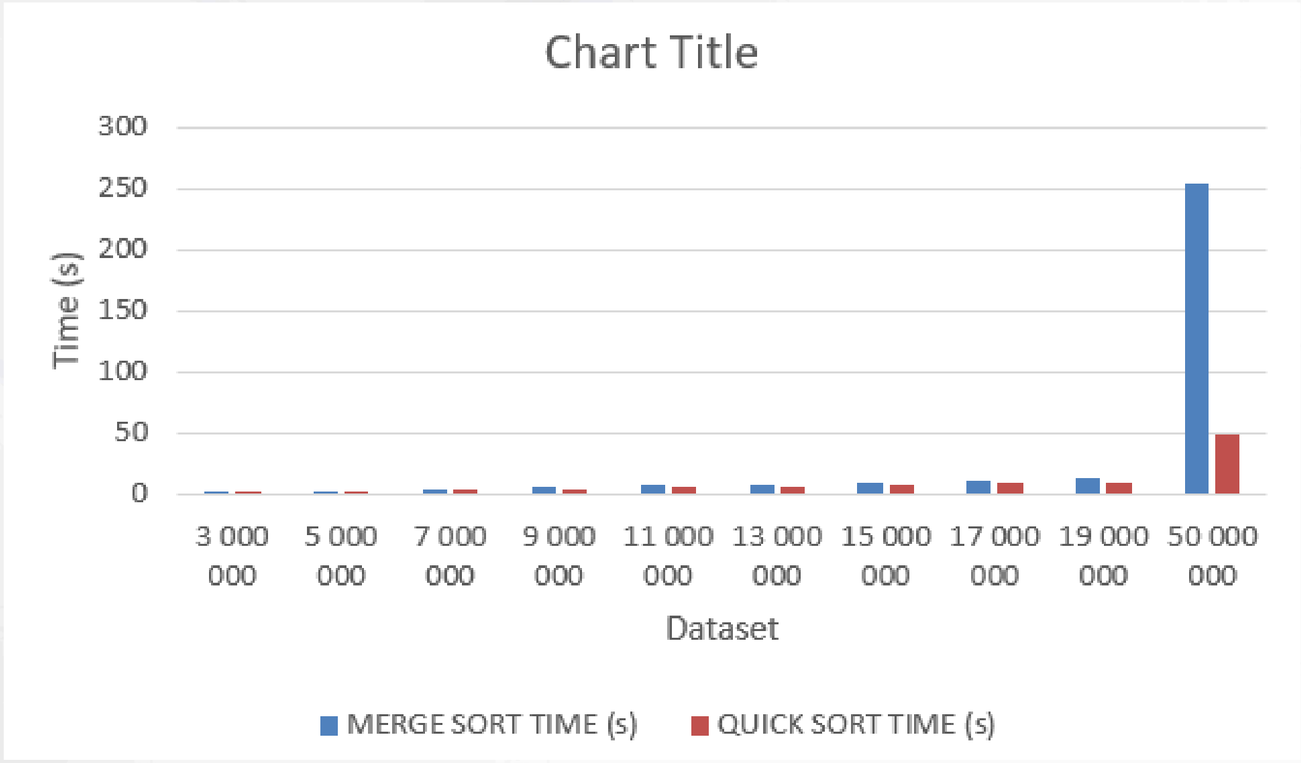
NO	DATASET	MERGE SORT TIME (s)	QUICK SORT TIME (s)
1	2 000 000	5.7381	4.9187
2	3 000 000	8.6953	7.6777
3	4 000 000	12.5153	11.7631
4	5 000 000	15.8529	14.3528
5	6 000 000	19.9691	19.2390
6	7 000 000	23.8725	23.5600
7	8 000 000	26.4044	25.6977
8	9 000 000	31.1411	28.8326
9	10 000 000	34.1187	32.1404
10	100 000 000	466.4188	526.1754



SORTING RUNTIME COMPARISON (JAVA)

AMIRUL

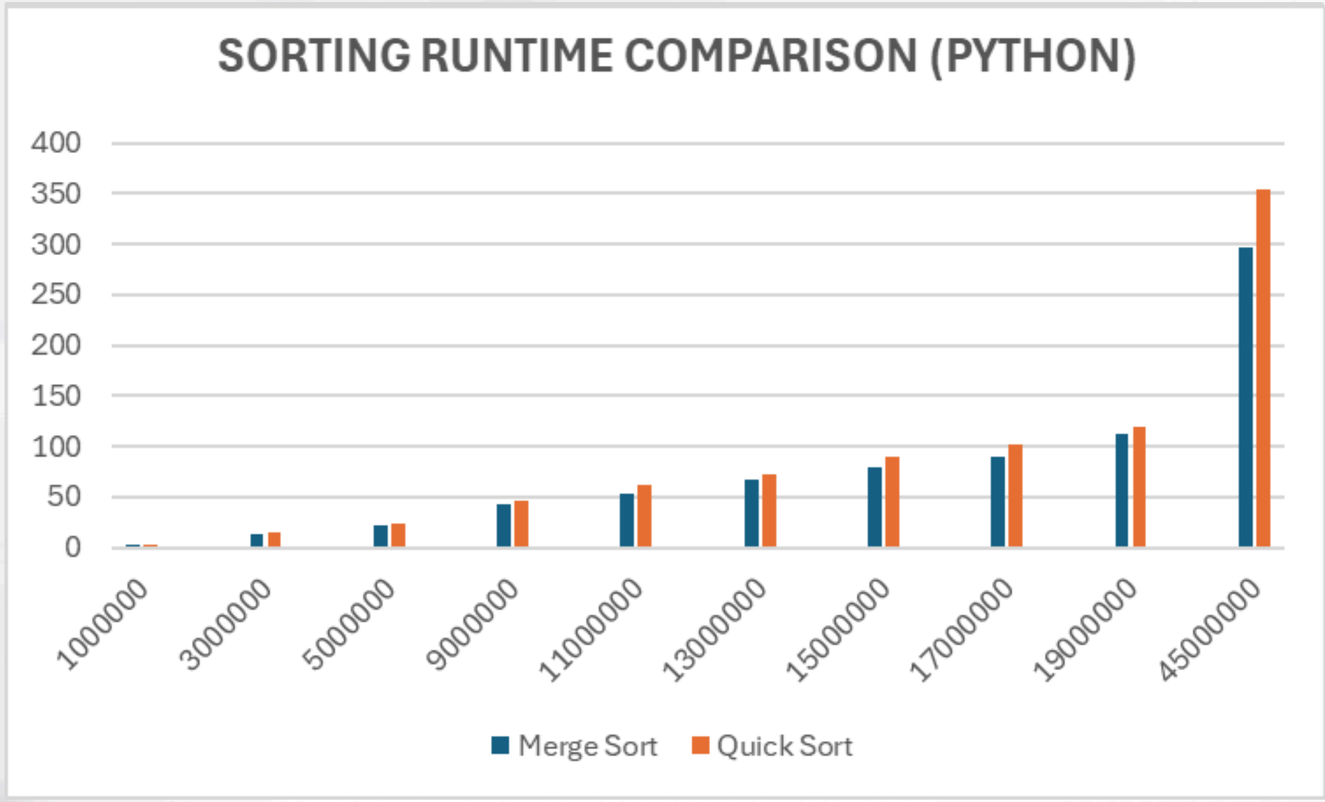
NO	DATASET	MERGE SORT TIME (s)	QUICK SORT TIME (s)
1	3 000 000	1.719	1.278
2	5 000 000	2.677	2.572
3	7 000 000	4.293	2.828
4	9 000 000	5.640	3.707
5	11 000 000	7.146	5.168
6	13 000 000	7.883	6.082
7	15 000 000	8.993	6.760
8	17 000 000	11.292	8.596
9	19 000 000	12.825	9.520
10	50 000 000	253.429	48.660



SORTING RUNTIME COMPARISON (PYTHON)

AZZRUDDIN

NO	DATASET	MERGE SORT TIME (s)	QUICK SORT TIME (s)
1	1000000	3.4653	3.3966
2	3000000	12.2566	14.7665
3	5000000	21.6394	24.3226
4	9000000	42.8352	46.1439
5	11000000	53.5350	61.6664
6	13000000	67.2718	71.9469
7	15000000	78.5289	89.6338
8	17000000	89.7913	102.1742
9	19000000	112.8287	118.6192
10	45000000	297.5753	354.1399



SEARCHING ANALYSIS RESULTS (BINARY SEARCH)

IQBAL (JAVA)

INPUT SIZE	USED FILE	WORST (ms)	AVERAGE (ms)	BEST (ms)
1	merge_sort_ 10000000	279.63	259.95	261.71
2	merge_sort_ 20000000	579.87	546.08	549.42

AMIRUL (JAVA)

INPUT SIZE	USED FILE	WORST (ms)	AVERAGE (ms)	BEST (ms)
1	merge_sort_3000000	194.49	177.64	139.74
2	merge_sort_19000000	1275.82	1207.66	866.14

AMEERUL (PYTHON)

INPUT SIZE	USED FILE	WORST (ms)	AVERAGE (ms)	BEST (ms)
1	merge_sort _5000000	9656.98	9154.92	9045.35
2	merge_sort _10000000	21219.61	20084.75	20890.87

AZZRUDDIN (PYTHON)

INPUT SIZE	USED FILE	WORST (ms)	AVERAGE (ms)	BEST (ms)
1	merge_sort_1000000	4503.18	4305.29	4239.13
2	merge_sort_3000000	15250.95	14328.55	9310.56

OUTPUT SAMPLES

OUTPUT SAMPLE (merge_sort_10.csv)

GENERATED DATASET

dataset_10.csv > data

```
1   6464374283,loxxrk
2   5047241965,ojxxsk
3   8929207004,dljfdw
4   8166397257,aorkax
5   772066069,vsqrhy
6   6327445370,valooe
7   9491841707,cxjerh
8   4682922225,krfwvt
9   3389584444,otlrrc
10  6169239128,wvzxwh
11
```

SORTED DATASET

merge_sort_10.csv > data

```
1   772066069,vsqrhy
2   3389584444,otlrrc
3   4682922225,krfwvt
4   5047241965,ojxxsk
5   6169239128,wvzxwh
6   6327445370,valooe
7   6464374283,loxxrk
8   8166397257,aorkax
9   8929207004,dljfdw
10  9491841707,cxjerh
```

OUTPUT

```
Merge Sort completed.
Input: dataset_10.csv
Output: merge_sort_10.csv
Running time: 0.000 s
```

OUTPUT SAMPLE (merge_sort_step_1_5.csv)

GENERATED DATASET

dataset_10.csv > data

```
1   6464374283,loxxrk
2   5047241965,ojxxsk
3   8929207004,dljfdw
4   8166397257,aorkax
5   772066069,vsqrhy
6   6327445370,valooe
7   9491841707,cxjerh
8   4682922225,krfwvt
9   3389584444,otlrrc
10  6169239128,wvzxwh
```

SORTED OUTPUT

merge_sort_step_1_5.txt

```
1  [6464374283/loxxrk, 5047241965/ojxxsk, 8929207004/dljfdw, 8166397257/aorkax, 772066069/vsqrhy]
2  [5047241965/ojxxsk, 6464374283/loxxrk, 8929207004/dljfdw, 8166397257/aorkax, 772066069/vsqrhy]
3  [5047241965/ojxxsk, 6464374283/loxxrk, 8929207004/dljfdw, 8166397257/aorkax, 772066069/vsqrhy]
4  [5047241965/ojxxsk, 6464374283/loxxrk, 8929207004/dljfdw, 772066069/vsqrhy, 8166397257/aorkax]
5  [772066069/vsqrhy, 5047241965/ojxxsk, 6464374283/loxxrk, 8166397257/aorkax, 8929207004/dljfdw]
6
```

OUTPUT SAMPLE (merge_sort_10.csv)

GENERATED DATASET

```
1   7157565882,xbxuat
2   942162917,xuzcok
3   1951974868,kjkvtl
4   5069615478,olpbox
5   4529166343,iosxdk
6   1115442276,afyidh
7   5209325226,fbhibw
8   5858384526,jlkukb
9   62265513,xulmcr
10  5848632093,akzefa
```

SORTED DATASET

```
1   62265513,xulmcr
2   942162917,xuzcok
3   1115442276,afyidh
4   1951974868,kjkvtl
5   4529166343,iosxdk
6   5069615478,olpbox
7   5209325226,fbhibw
8   5848632093,akzefa
9   5858384526,jlkukb
10  7157565882,xbxuat
```

OUTPUT

```
Enter CSV filename : dataset_10.csv
-Sorted 10 records.
-Time taken (excluding I/O): 0.0000 seconds.
-Output file name: merge_sort_10.csv
```

OUTPUT SAMPLE (merge_sort_step_1_5.csv)

GENERATED DATASET

```
7157565882,xbxuat
942162917,xuzcok
1951974868,kjkvtl
5069615478,olpbox
4529166343,iosxdk
1115442276,afyidh
5209325226,fbhibw
5858384526,jlkukb
62265513,xulmcr
5848632093,akzefa
```

SORTED OUTPUT

```
[7157565882/xbxuat, 942162917/xuzcok, 1951974868/kjkvtl, 5069615478/olpbox, 4529166343/iosxdk]
[942162917/xuzcok, 7157565882/xbxuat, 1951974868/kjkvtl, 5069615478/olpbox, 4529166343/iosxdk]
[942162917/xuzcok, 1951974868/kjkvtl, 7157565882/xbxuat, 5069615478/olpbox, 4529166343/iosxdk]
[942162917/xuzcok, 1951974868/kjkvtl, 7157565882/xbxuat, 4529166343/iosxdk, 5069615478/olpbox]
[942162917/xuzcok, 1951974868/kjkvtl, 4529166343/iosxdk, 5069615478/olpbox, 7157565882/xbxuat]
```

OUTPUT SAMPLE (quick_sort_10.csv)

GENERATED DATASET

dataset_10.csv

```
1    2785334288,jbuzpk
2    2604811430,nomrxh
3    5162709560,wxrcxv
4    515319273,vyppei
5    7939752666,zqtvbj
6    1871561299,rhptec
7    9204602944,nrpgst
8    4943344457,sqybhw
9    2979643268,riombi
10   4986158329,xnsgne
```

SORTED DATASET

quick_sort_10.csv

```
1    515319273,vyppei
2    1871561299,rhptec
3    2604811430,nomrxh
4    2785334288,jbuzpk
5    2979643268,riombi
6    4943344457,sqybhw
7    4986158329,xnsgne
8    5162709560,wxrcxv
9    7939752666,zqtvbj
10   9204602944,nrpgst
```

OUTPUT

```
PS C:\Users\ameer\Downloads\data> java quick_sort dataset_10.csv
Quick Sort completed.
Input: dataset_10.csv
Output: quick_sort_10.csv
Runtime: 0.000 s
```

OUTPUT SAMPLE (quick_sort_step_1_5.csv)

GENERATED DATASET

dataset_10.csv

```
1    2785334288,jbuzpk
2    2604811430,nomrxh
3    5162709560,wxrcxv
4    515319273,vyppei
5    7939752666,zqtvbj
6    1871561299,rhptec
7    9204602944,nrpgst
8    4943344457,sqybhw
9    2979643268,riombi
10   4986158329,xnsgne
```

SORTED OUTPUT

```
1   [2604811430/nomrxh, 5162709560/wxrcxv, 515319273/vyppei, 7939752666/zqtvbj, 1871561299/rhptec]
2   pi=1 [515319273/vyppei, 1871561299/rhptec, 2604811430/nomrxh, 7939752666/zqtvbj, 5162709560/wxrcxv]
3   pi=3 [515319273/vyppei, 1871561299/rhptec, 2604811430/nomrxh, 5162709560/wxrcxv, 7939752666/zqtvbj]
```

OUTPUT SAMPLE (quick_sort_10.csv)

GENERATED DATASET

dataset_10.csv

```
1    8883026663,dgulbl
2    8032474628,hjwuze
3    2628079375,rbewsg
4    7669434732,lolcfm
5    255834513,iqbfzj
6    7468591491,mxzsny
7    203001867,aphiwt
8    2475751053,rjbduv
9    783925171,wjqigy
10   1590521040,aavyng
```

SORTED DATASET

quick_sort_10.csv

```
1    203001867,aphiwt
2    255834513,iqbfzj
3    783925171,wjqigy
4    1590521040,aavyng
5    2475751053,rjbduv
6    2628079375,rbewsg
7    7468591491,mxzsny
8    7669434732,lolcfm
9    8032474628,hjwuze
10   8883026663,dgulbl
```

OUTPUT

```
PS C:\Users\user\Desktop\Algo> python quick_sort.py
Enter dataset filename (e.g., dataset_1000.csv): dataset_10.csv
Sorting 10 records using Quick Sort...
Sorted 10 records.
Runtime: 0.0000 seconds
Output file: quick_sort_10.csv
```

OUTPUT SAMPLE (quick_sort_step_1_5.csv)

GENERATED DATASET

dataset_10.csv

```
1    8883026663,dgulbl
2    8032474628,hjwuze
3    2628079375,rbewsg
4    7669434732,lolcfm
5    255834513,iqbfzj
6    7468591491,mxzsny
7    203001867,aphiwt
8    2475751053,rjbduv
9    783925171,wjqigy
10   1590521040,aavyng
```

SORTED OUTPUT

quick_sort_step_1_5.txt

```
1   [8032474628/hjwuze, 2628079375/rbewsg, 7669434732/lolcfm, 255834513/iqbfzj, 7468591491/mxzsny]
2   pi=2 [2628079375/rbewsg, 255834513/iqbfzj, 7468591491/mxzsny, 8032474628/hjwuze, 7669434732/lolcfm]
3   pi=0 [255834513/iqbfzj, 2628079375/rbewsg, 7468591491/mxzsny, 8032474628/hjwuze, 7669434732/lolcfm]
4   pi=3 [255834513/iqbfzj, 2628079375/rbewsg, 7468591491/mxzsny, 7669434732/lolcfm, 8032474628/hjwuze]
5
```

OUTPUT SAMPLE (BinarySearchStep)

TARGET NOT FOUND

binary_search_step_70.txt

```
499999: 4997981498/bslvpm
249999: 2498329748/wlfocl
124999: 1252545899/qqonpg
62499: 627728181/exegpb
31249: 314775206/evtuki
15624: 157961086/mxvuvf
7811: 78959331/omiddr
3905: 39530018/njndfn
1952: 20122380/ygybjb
975: 9844224/kzrwnz
487: 4922613/zstyma
243: 2407012/ztfsod
121: 1050564/lmrxfd
60: 449403/prbchc
29: 222544/acnouv
14: 97141/vtvngm
6: 22377/sappzm
2: 15022/svnhii
0: 3184/adnldw
-1
```

TARGET FOUND

binary_search_step_53982.txt

```
499999: 4997981498/bslvpm
249999: 2498329748/wlfocl
124999: 1252545899/qqonpg
62499: 627728181/exegpb
31249: 314775206/evtuki
15624: 157961086/mxvuvf
7811: 78959331/omiddr
3905: 39530018/njndfn
1952: 20122380/ygybjb
975: 9844224/kzrwnz
487: 4922613/zstyma
243: 2407012/ztfsod
121: 1050564/lmrxfd
60: 449403/prbchc
29: 222544/acnouv
14: 97141/vtvngm
6: 22377/sappzm
10: 53982/boehwf
```

OUTPUT SAMPLE (binary_search_step)

TARGET NOT FOUND

binary_search_step_30.txt

```
999999: 4994403350/vfvxcp
499999: 2498138715/krgxqz
249999: 1248246556/kgyavi
124999: 626296575/oidtip
62499: 311023687/rnffdb
31249: 155477896/xtrafn
15624: 78328617/tosyxz
7811: 39046337/pvkfzi
3905: 19281005/xoatlf
1952: 9527536/enkqyd
975: 4822527/mmklmw
487: 2503470/gxksgt
243: 1205445/icjevm
121: 508162/bqtnab
60: 238815/pbwmhu
29: 116295/kyueel
14: 54827/hurloi
6: 34130/bdtwgq
2: 12818/ydejcr
0: 1557/xzzovg
-1

```

TARGET FOUND

binary_search_step_4446161425.txt

```
999999: 4994403350/vfvxcp
499999: 2498138715/krgxqz
749999: 3747776497/hdmdpo
874999: 4370748185/koqoad
937499: 4680095253/cigekb
906249: 4525573982/blycns
890624: 4447536439/zyfvuo
882811: 4409186165/qioicl
886717: 4428109042/hkltjd
888670: 4437963925/oeeaas
889647: 4442749452/pafibt
890135: 4445215768/aptrya
890379: 4446408882/krbtlc
890257: 4445857538/exvmot
890318: 4446147922/tvbtqy
890348: 4446282706/ufglni
890333: 4446227500/izkzpx
890325: 4446175205/ywejfz
890321: 4446156228/rmifhr
890323: 4446161425/gwtcdc

```

CONCLUSIONS

Analysis Findings

Same Language, Same Hardware

- In both Python and Java, Merge Sort generally offered more stable and predictable performance.
- Quick Sort was faster than Merge Sort because the algorithm sorts in-place which does not require extra memory for another array
- Binary Search executed rapidly in all cases due to its logarithmic time complexity, with minimal difference across test cases.

Same Language, Different Hardware

- Hardware specifications significantly affected performance:
- Faster CPUs and more RAM reduced execution time for all algorithms.
- Merge Sort benefited from better memory, while Quick Sort performance was sensitive to CPU and stack capacity.

Different Languages, Same Hardware

- Java outperformed Python in most cases, especially on large datasets, due to its compiled nature and efficient memory management.
- Python, being an interpreted language, showed slower runtimes, particularly for recursive functions like Quick Sort and Merge Sort.
- Python have recursion limit which cause Quick Sort in Python to struggle with larger dataset.

Best Sorting Arrays

To efficiently simulate AVL trees using arrays, the data must first be sorted, as AVL operations rely on balanced and sorted structures. The choice of sorting algorithm directly affects both preprocessing time and space usage especially when working with large datasets.

Merge Sort

Merge Sort provides stable **O(n log n)** performance in all scenarios. It makes sure sorted output even for worst-case data, making it highly reliable for AVL tree construction. Its divide-and-conquer strategy works well even with ordered or patterned input. However, Merge Sort requires O(n) additional memory, which can be a drawback for very large datasets or memory-constrained environments.

Quick Sort

Quick Sort has a theoretical **$O(n^2)$** worst-case time, it remains one of the fastest sorting algorithms in practice especially for large datasets. Despite using a suboptimal pivot (last element) Quick Sort still outperformed Merge Sort in our tests when working making it more space-efficient, and its performance improves significantly with large-scale data.

While Merge Sort is more stable and safer against worst-case input, Quick Sort is ultimately the faster option in practice even when using the last-element pivot.

AVL vs Linked Structure

Performance & Efficiency

We use arrays because they have constant time (O(1) time) access via indexing, which is optimal for fast lookups as we see in our binary search2. py.

Linked structures (O(n) for reaching the elements) also use more space because of pointer fields.

Simplicity in Coding

In our sorting implementations (merge_sort. py, quick_sort. py, using array improve the logic and prevent bugs.

Linked lists would on the other hand require both node creation, pointer logic and balacing (for AVL) which would stall development time due to added complexity and error prone code.

Practical Relevance in Sorting & Searching

We were able to get log search time on sorted anays with our Binary Search.

Both Merge Sort and Quick Sort require random access to elements, which is not possible in linked structures.