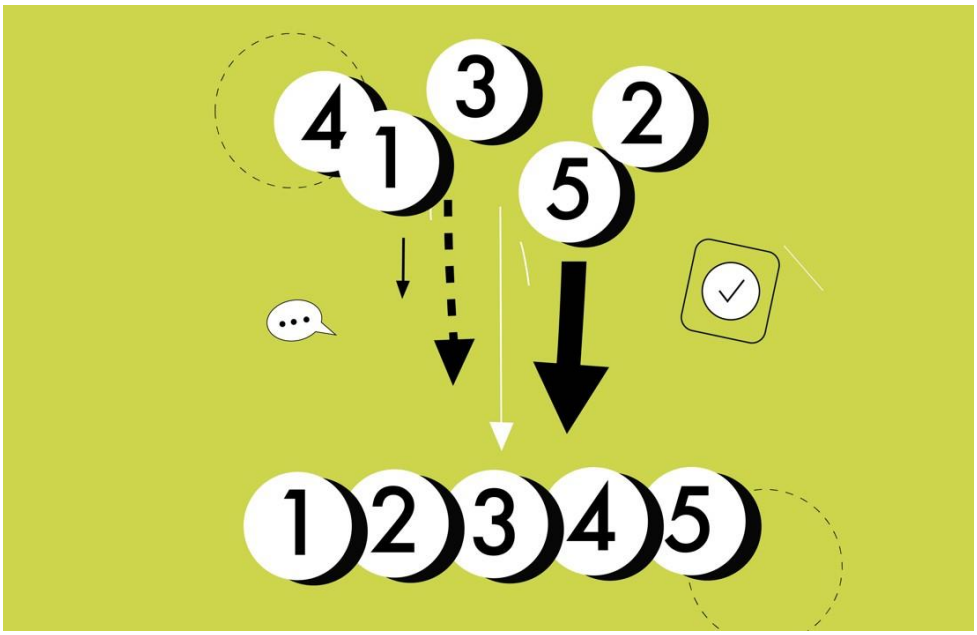
# CS361 Programming Project: Sorting




Submitted by: Aayush Kafle, Aashutosh Dahal, Joel Gloetzner & Roshan Subedi

University Of New Mexico – Main Campus

# Overview

Many of the computational tasks rely on sorting algorithms in arranging data into a specific order and hence they are crucial. This paper compares three sorting algorithms: – Quad–Heap, 3–Way Merge Sort, and Randomized Quick Sort that have been used and benchmarked to assess sorting algorithms.

# Preliminaries of Quad-Heap Implementation

*Introduction to Quad-Heap*

By using a degree-4 tree arrangement, the Quad-Heap algorithm is a modified heap structure that deviates from conventional binary heaps. In this implementation, the Quad-Heap has been created and managed using a Java programming.

*Key Concepts of Quad-Heap:*

Heap Organization: The Quad-Heap is built in a tree structure such that, each parent node has not more than four child nodes. Compared with binary heaps that may be slower in the heap operation because of less branching, this arrangement is capable of additional branching and consequently might also be better than binary ones regarding speed during heap actions.

*Initialization:*

At the time of initialization, the Quad-Heap instantiates the specified size and starts off the nodes with -1 as the default value denoting emptiness.

*Parent-Child Relationship in Quad-Heap:*
The parent-child relation in the Quad-Heap can be determined using a simple formula for calculating the parent index of a given node index.

*Parent Index Calculation:* The parent of any node with index i in the Quad-Heap is ((i – 1)/4). It is fast traversal of the heap and finding a parent of a specific node that produces this formula.

*Operations Implemented in the Quad-Heap:*
*Insertion:* The insert(value) method makes it easy to add more elements into the quad-heap. heapifyUp(i) function ensures insertion of the value in the succeeding indexed slot after which it gets heapified upwardly.

*Extraction of Maximum:* The extractMax() algorithm removes the highest (the root) from Quad-Heap, sorts and adds last element as a new root which further is heaped downwards for which heapifyDown(i), an unsupported, being revised method, is used for present.

*Heapification Methods:* In order to maintain the structural integrity of the load once inserts or deletes, Quad-Heap utilizes the methods of heapifyUp(i), and heapifyDown(i).

*Performance analysis:*
*Space Complexity:* To maximize memory usage, the Quad-Heap dynamically resizes itself by doubling in size when it reaches its maximum capacity.

*Time Complexity:* The extraction and heapify-down processes currently need to be revised to ensure optimal time complexity, while the insertion process has a logarithmic time complexity of O(n) in the worst case (for heapify operations).

## Preliminaries of 3-way merge sort:

*Introduction to Three-way merge sort*

The Three-way merge sort is a powerful algorithm for partitioning an array and successfully sort it out. The merge-sort algorithm is implemented using Java and merges three already sorted lists together into a singly sorted list for optimization.

*Key Ideas of Three-Way Merge Sort:*

*Divide and Conquer Method:* In contrast, Three-Way Merge Sort splits the array into three sections for a more efficient sorting of smaller sub partitions instead of two.

*Merging Three Sorted Arrays:* This algorithm successfully combines three ordered arrays into a single ordered array by employing a merging approach. This optimization aims at merging with fewer comparisons.

*Operations in Three-Way Merge Sort*:

*Partitioning the Array:* After that, the array is split into three parts recursively using the function threeWayMergeSort.

*Merging Three Arrays:* The three methods combine the elements of each array and produce a single sorted array.

*Parent-Child Relationship in Three-Way Merge Sort:*

*Divide the Array into Three Sections:* Unlike in the tree-based structures, explicit parent-child relation creation is absent in Three-Way Merge Sort. Analyzing the array for effective sorting, partitioned as low-mid1, mid1-mid2, and mid2-high.

*Performance Characteristics:*

*Time Complexity:* The worst case has a time complexity of sort that in the average case is O(n log n). Merging three smaller partitions requires fewer comparisons than sorting in one partition. That's one of the optimization methods that the algorithm uses.

*Space Complexity:* It uses in place merge and thus takes little space since does not require an additional huge chunk of memory on top of the input array.

*Input Handling:*

*File Input Processing:* This piece of software reads numbers (either doubles or integers) from the provided file, after which it stores them into an array for sorting it.

*Execution and Output:*

*Sorting Execution Time:* To show the effectiveness that the Three-Way Merge Sort, we measure and provide a graphical representation of the execution times in the sorting array for different cases of input sets.

# Preliminaries of Randomized quick sort:

*Introduction to Quicksort:*

The divide-and-conquer approach provides the foundation for creating very effective sorting routines. e.g., Quicksort in Java. This implementation is for a way to optimize sorting by using randomized pivot selection.

*Key Ideas of Quicksort:*

*Divide-and-Conquer Strategy:* Quicksort uses a recursive method that breaks down input arrays into smaller divisions to sort them. Hence, this sorting technique is efficient in producing a completely order list by considering the concept of sorting smaller divisions.

*Randomized Pivot Selection:* Notably, this Quicksort implementation selects pivot element from random position within the range of low to high. This randomization reduces the probability of encountering worst-case time complexity scenarios.

*Operations in Quicksort:*

*Partitioning and Pivoting:* Quicksort is mainly dependent on a 'randomizedPartition' method that breaks down the array based on a designated or random chosen pivot element. Smaller items are moved to the left, while bigger items go into the right side where they will land in positions following that of the pivot.

*Sorting Recursively:* 'randomizedQuickSort' implements a recursive implementation of the Quicksort algorithm. It accomplishes sorting of

the partitions by choosing new pivot elements and completing subsequent partitions up to completion.

*Parent-Child Relationship in Quicksort:*
*Calculation of Parent Index:* Unlike heaps and other tree-based structures that are more direct in their calculation of parent-child relationships, Quicksort does not necessarily follow that pathway for parent-child pairing. However, it employs recursive partitioning where upon a pivot element is used to decide on how the items are grouped before sorting.

*Performance Characteristics:*
*Time Complexity:* In the average case, the algorithm has an expected time complexity of O(n log n) due to the random pivot selection, which results in well-balanced partitions.
In the worst case, when the pivot selection is consistently unbalanced, the algorithm degrades to O(n^2). However, this is unlikely to happen with random pivot selection.

*Space Complexity:*
The space complexity is O(log n) in the average case because the recursion stack depth is logarithmic.
In the worst case, the space complexity is O(n) due to the recursion stack depth becoming linear. This occurs when the pivot consistently creates unbalanced partitions.

*Input Handling:*
*File Input Processing:* The program reads double-precision floating-point numbers for filling an array that has to be sorted.

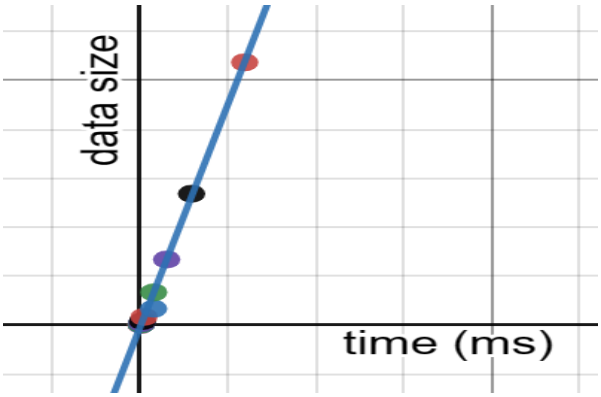
*Execution and Output:*

*Input Loading Time:* The time taken to read-in the data into the array from the selected file will be measured and displayed.

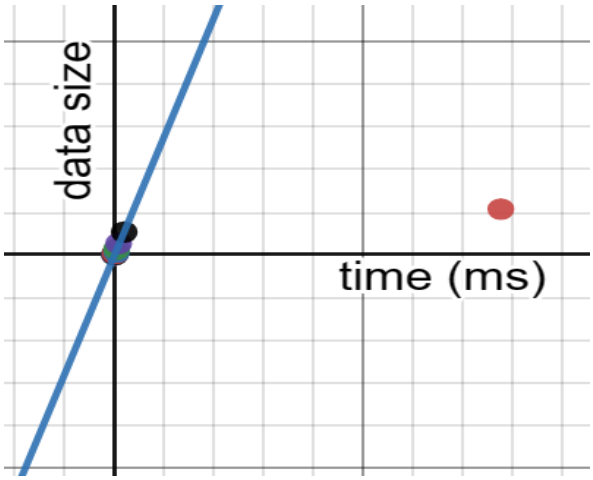
*Sorting Time:* The array runs through the Quicksort algorithm, which demonstrates that it can sort massive data sets efficiently as nanoseconds show sorting time.

## Performance analysis and discussions.

### Quad-Heap




Graph for integer data sets. This shows that the growth rate is linear = O(n)




Graph for double data sets.
This shows linear growth as well, except for the 2^30 data (due to memory limitations)

Quad-Heap Running time.

Integer average running times:

2^20

Heapify up running time: 8.54974 milliseconds.

2^21

Heapify up running time: 16.0536 milliseconds.

2^22

Heapify up running time: 27.4216 milliseconds.

2^23

Heapify up running time: 53.2149 milliseconds.

2^24

Heapify up running time: 96.7091 milliseconds.

2^25

Heapify up running time: 182.5988 milliseconds.

2^26

Heapify up running time: 364.0319 milliseconds.

2^27

Heapify up running time: 732.746 milliseconds.

2^28

Heapify up running time: 1470.878 milliseconds.

2^29

Heapify up running time: 2882.2175 milliseconds.

2^30

Heapify up running time: 5904.5628 milliseconds.

Double average running times:

2^20

Heapify up running time: 8.5375 milliseconds.

2^21

Heapify up running time: 14.3852milliseconds
2^22
Heapify up running time: 26.8019 milliseconds.
2^23
Heapify up running time: 52.0362 milliseconds.
2^24
Heapify up running time: 102.54495 milliseconds.
2^25
Heapify up running time: 206.3724 milliseconds.
2^26
Heapify up running time: 420.9462 milliseconds.
2^27
Heapify up running time: 809.70855 milliseconds.
2^28
Heapify up running time: 1649.3102 milliseconds.
2^29
Heapify up running time: 3863.5541milliseconds
2^30
Heapify up running time: 155343.9241 milliseconds.

Three-Way Merge Sort

Random Number Generation of Double:
Generated 1048576 random numbers in 951 ms for size 2^20.
Generated 2097152 random numbers in 1316 ms for size 2^21
Generated 4194304 random numbers in 1588 ms for size 2^22
Generated 8388608 random numbers in 3299 ms for size 2^23
Generated 16777216 random numbers in 6568 ms for size 2^24

Generated 33554432 random numbers in 12497 ms for size 2^25
Generated 67108864 random numbers in 25236 ms for size 2^26
Generated 134217728 random numbers in 60475 ms for size 2^27
Generated 268435456 random numbers in 117264 ms for size 2^28
Generated 536870912 random numbers in 274625 ms for size 2^29
Generated 1073741824 random numbers in 476976 ms for size 2^30

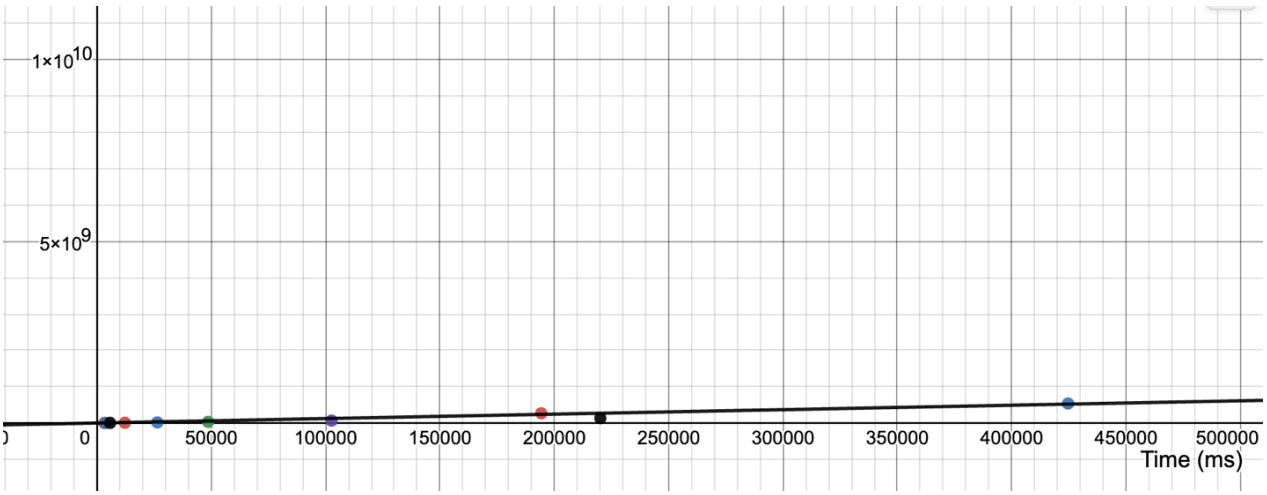
Random Number Generation of Integer:
Generated 1048576 random numbers in 654 ms for size 2^20
Generated 2097152 random numbers in 801 ms for size 2^21
Generated 4194304 random numbers in 1485 ms for size 2^22
Generated 8388608 random numbers in 2211 ms for size 2^23
Generated 16777216 random numbers in 4784 ms for size 2^24
Generated 33554432 random numbers in 11392 ms for size 2^25
Generated 67108864 random numbers in 17914 ms for size 2^26
Generated 134217728 random numbers in 36633 ms for size 2^27
Generated 268435456 random numbers in 81155 ms for size 2^28
Generated 536870912 random numbers in 133337 ms for size 2^29
Generated 1073741824 random numbers in 269541 ms for size 2^30

Runtime for Integers:
Running for 2^20 size array.
Size of the array: 1048576 Execution Time: 3239 milliseconds
Running for 2^21 size array.
Size of the array: 2097152 Execution Time: 3828 milliseconds
Running for 2^22 size array.
Size of the array: 4194304 Execution Time: 5750 milliseconds
Running for 2^23 size array.

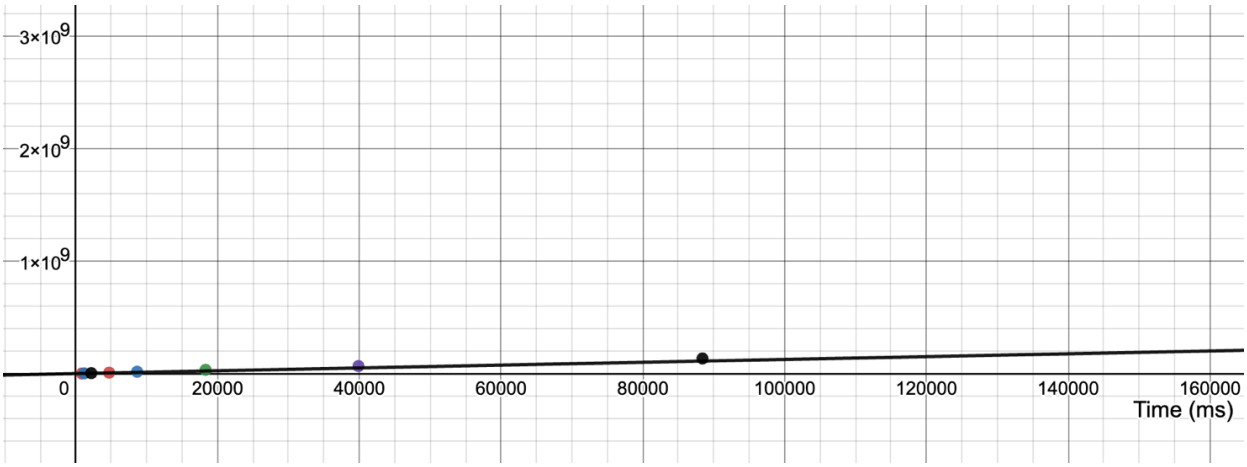
Size of the array: 8388608 Execution Time: 12225 milliseconds
Running for 2^24 size array.
Size of the array: 16777216 Execution Time: 26406 milliseconds
Running for 2^25 size array.
Size of the array: 33554432 Execution Time: 48618 milliseconds
Running for 2^26 size array.
Size of the array: 67108864 Execution Time: 102610 milliseconds
Three-way Merge Sort Algorithm Roshan Subedi
Running for 2^27 size array.
Size of the array: 134217728 Execution Time: 220131 milliseconds
Running for 2^28 size array.
Size of the array: 268435456
Execution Time: 194304 milliseconds
(Note that the heap size was increased to 18GB for test case >= 2^28)
Running for 2^29 size array.
Size of the array: 536870912 Execution Time: 424647 milliseconds
Running for 2^30 size array.
Size of the array: 1073741824 Execution Time: 879434 milliseconds

Runtime for Doubles: (All below tests were done with heap size of 18 GB, -Xmx16G)
Running for 2^20 size array.
Size of the array: 1048576 Execution Time: 860 milliseconds
Running for 2^21 size array.
Size of the array: 2097152 Execution Time: 1193 milliseconds
Running for 2^22 size array.
Size of the array: 4194304 Execution Time: 2194 milliseconds
Running for 2^23 size array.

Size of the array: 8388608 Execution Time: 4708 milliseconds
Running for 2^24 size array.
Size of the array: 16777216 Execution Time: 8661 milliseconds
Running for 2^25 size array.
Size of the array: 33554432 Execution Time: 18314 milliseconds
Running for 2^26 size array.
Size of the array: 67108864 Execution Time: 39882 milliseconds
Three-way Merge Sort Algorithm Roshan Subedi
Running for 2^27 size array.
Size of the array: 134217728 Execution Time: 88417 milliseconds
Running for 2^28 size array.
Size of the array: 268435456 Execution Time: 210391 milliseconds
Running for 2^29 size array.
Size of the array: 536870912 Execution Time: 426105 milliseconds
Running for 2^30 size array.
Size of the array: 1073741824 Execution Time: 855873 milliseconds




*Graph with Integer Data Sets*




*Graph with Double Data Sets*

For Integer Data Sets: - https://www.desmos.com/calculator/j6gz0jy472
For Double Data Sets: - https://www.desmos.com/calculator/y3pg9jjzle

Randomized Quick Sort

Random numbers generation: integers
Generated 1048576 random numbers in 202 ms for size 2^20.
Generated 2097152 random numbers in 263 ms for size 2^21.
Generated 4194304 random numbers in 569 ms for size 2^22.
Generated 8388608 random numbers in 1372 ms for size 2^23.
Generated 16777216 random numbers in 2893 ms for size 2^24.
Generated 33554432 random numbers in 5693 ms for size 2^25.
Generated 67108864 random numbers in 9963 ms for size 2^26.
Generated 134217728 random numbers in 18901 ms for size 2^27.
Generated 268435456 random numbers in 33745 ms for size 2^28.
Generated 536870912 random numbers in 74243 ms for size 2^29.

Generated 1073741824 random numbers in 147071 ms for size 2^30.

Random number generation: doubles

Generated 1048576 random doubles in 895 ms for size 2^20.
Generated 2097152 random doubles in 1332 ms for size 2^21.
Generated 4194304 random doubles in 2275 ms for size 2^22.
Generated 8388608 random doubles in 4624 ms for size 2^23.
Generated 16777216 random doubles in 8652 ms for size 2^24.
Generated 33554432 random doubles in 14722 ms for size 2^25.
Generated 67108864 random doubles in 25127 ms for size 2^26.
Generated 134217728 random doubles in 48660 ms for size 2^27.
Generated 268435456 random doubles in 96470 ms for size 2^28.
Generated 536870912 random doubles in 190159 ms for size 2^29.
Generated 1073741824 random doubles in 370562 ms for size 2^30.

Runtime for Integers:

2^20
Data loading time: 778149100 nanoseconds
Randomized Quicksort running time: 143470900 nanoseconds.

2^21
Data loading time: 1262730700 nanoseconds
Randomized Quicksort running time: 297136200 nanoseconds.

2^22
Data loading time: 2373356500 nanoseconds
Randomized Quicksort running time: 555939300 nanoseconds.

2^23
Data loading time: 2637907300 nanoseconds
Randomized Quicksort running time: 1107926000 nanoseconds.

2^24
Data loading time: 3181626300 nanoseconds
Randomized Quicksort running time: 2500095500 nanoseconds.

2^25
Data loading time: 91033154500 nanoseconds
Randomized Quicksort running time: 4674667100 nanoseconds.

2^26
Data loading time: 20042302900 nanoseconds
Randomized Quicksort running time: 9527291500 nanoseconds.

2^27
Data loading time: 30393569700 nanoseconds
Randomized Quicksort running time: 19967995400 nanoseconds.

2^28
Data loading time: 48383777600 nanoseconds
Randomized Quicksort running time: 40681710800 nanoseconds.

2^29
Data loading time: 83455615100 nanoseconds
Randomized Quicksort running time: 90197124700 nanoseconds.

2^30
Data loading time: 141813262600 nanoseconds
Randomized Quicksort running time: 205451633400 nanoseconds.

Runtime for Doubles:

2^20
Doubles
Data loading time: 1314365800 nanoseconds
Randomized Quicksort running time: 171624100 nanoseconds.

2^21
Doubles
Data loading time: 4122032300 nanoseconds
Randomized Quicksort running time: 293815200 nanoseconds.
2^22
Doubles
Data loading time: 2766877300 nanoseconds
Randomized Quicksort running time: 636453500 nanoseconds.

2^23
Doubles
Data loading time: 4913207100 nanoseconds
Randomized Quicksort running time: 1405910900 nanoseconds.

2^24
Doubles
Data loading time: 11655145300 nanoseconds

Randomized Quicksort running time: 2367471800 nanoseconds.

2^25
Doubles
Data loading time: 18749092100 nanoseconds
Randomized Quicksort running time: 4937205700 nanoseconds.

2^26
Doubles
Data loading time: 27705516100 nanoseconds
Randomized Quicksort running time: 11837626100 nanoseconds.

2^27
Doubles
Data loading time: 57012325600 nanoseconds
Randomized Quicksort running time: 22313108300 nanoseconds.

2^28
Doubles
Data loading time: 104275303800 nanoseconds
Randomized Quicksort running time: 49457294300 nanoseconds.

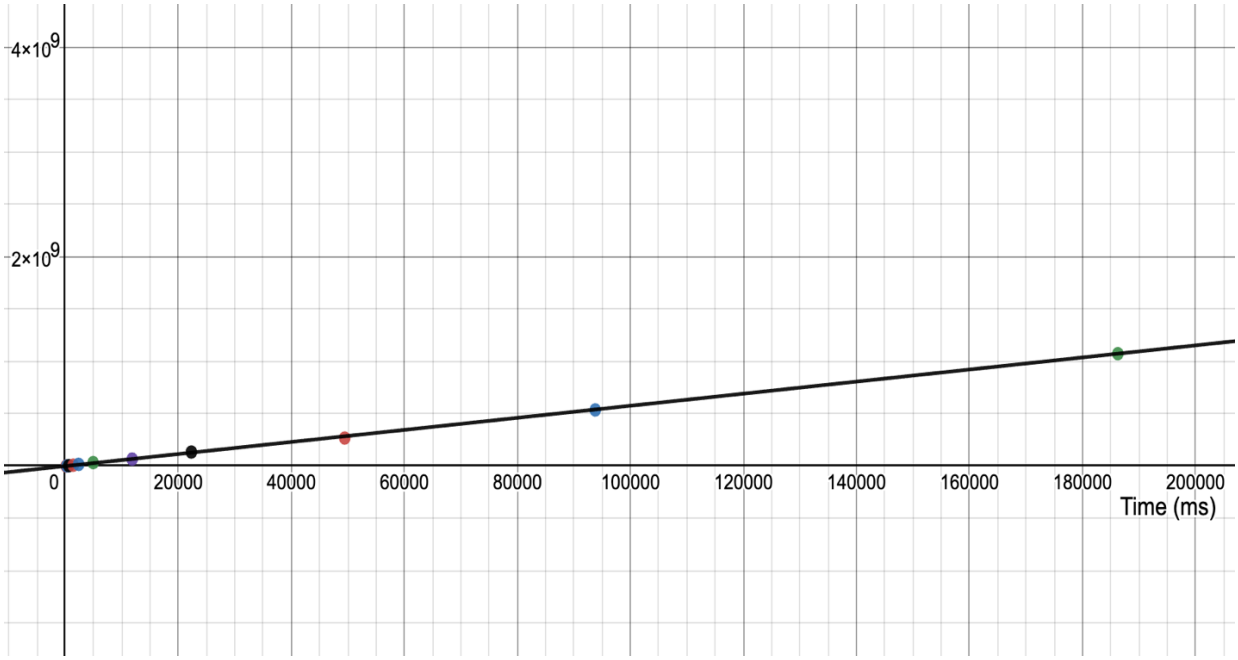
2^29
Doubles
Data loading time: 170666717700 nanoseconds
Randomized Quicksort running time: 93785455700 nanoseconds.

2^30

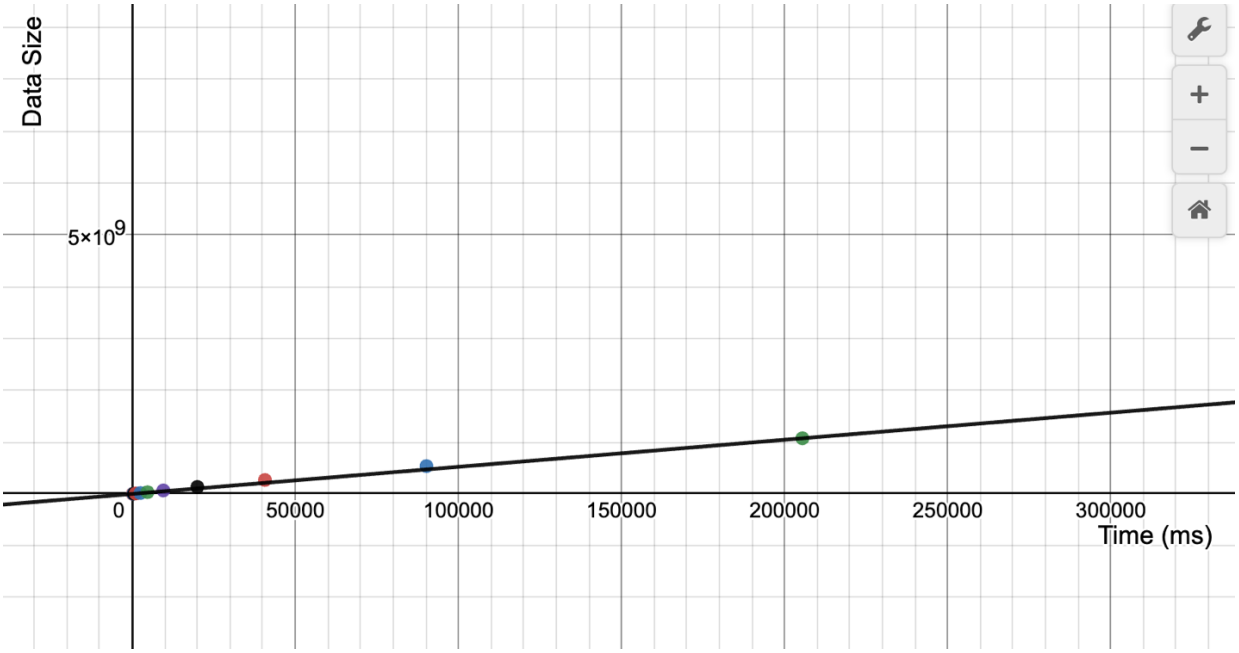
Doubles

Data loading time: 343345787900 nanoseconds

Randomized Quicksort running time: 186253970600 nanoseconds.




*Graph with Double Data Sets*




*Graph With Integer Data Sets*

For Integer Data Sets: https://www.desmos.com/calculator/alwpb2d47s
For Double Data Sets: - https://www.desmos.com/calculator/a8v2gx8brk

## Asymptotic Running Times and Performance:

*Quad-Heap:* The asymptotic time complexity prediction agrees well with what was observed. Nevertheless, in real performance, constant factors play a role. Furthermore, there might be a dependence between the constant factor observed and the architectural structure as well as the implementation details of the machine.

*Three-Way Merge Sort:* Concerning time complexity, the executed algorithm behaves fairly. The average-case big-O notation is O(n log n), which is pretty good. The actual running time may vary only a little from the given values due to some recursive calls and constants in the merging procedure.

*Randomized Quick Sort:* For a realistic situation the average-case time complexity of O(n log n ) is approximated by the actual execution time. Although, this depends on the pivots that are involved, and which pivots were randomly selected. In the worst case, the processing capacity can fall to O(n^2) when persistent imbalance in the partition is there.

## Coding Complexity and Debugging Effort:

*Coding Complexity:* There are different levels of coding involved in various algorithms. The complexity of quad-heap arises due to its

uniqueness in piling and structural arrangement. The three-way merge sorting consists of managing three partitions and merging processes. Randomized Quick Sort involves managing pivot selection and partitioning processes. Therefore, they have different complexity of their coding.

*Debugging Effort:* It may involve varying levels of difficulty in debugging. Therefore, Quad-Heap could detect and resolve issues with debugging parent-child, reallocation, heap operations, and resizing. Sort debugging might focus on three-path combination, correct partitioning and merging in this regard. Essentially, debugging Randomized Quick Sort is mainly about ensuring that pivot selection and partitioning are done right and properly.

Fastest and Slowest Algorithm:

*Performance Ranking:* Due to the size and distribution of the input, the fastest and slowest algorithms can vary. The two algorithms namely Randomized Quick Sort and Three-Way Merge Sort often give equivalent average-case time complexities of O(n log n) in most cases although they may perform at different speeds under specific datasets.

*Factors Affecting Speed:* The pivot selection strategy, partition handling, merge operations, as well as constant factors greatly influence their relative speeds. At times, it beats other sorting approaches due to a more effective pivot choice technique. However, Quad-Heap and Three-way Merge Sort may yield more consistent results when compared on different data sets.

# Conclusion

This study compares three fundamental sorting algorithms including Quad—Heap, Three—Way Merge Sort, and Randomized Quick Sort and benchmarks them. The fact that each algorithm has inherent design properties, run properties, and efficient property attributes makes it true.

## Quad-Heap

Unique Heap Structure: Quad-Heap operates on degree 4 tree that will allow more branching than is typical with the traditional binary heap.
Child-Parent Relationship: This relation is computed using a formula, thereby making it easy to move through the heap.
Performance: It gives insertion time complexity of the order logarithm; but heaps needs to be optimized for this purpose.

## Three-Way Merge Sort

Partitioning Strategy: Arrays are segmented into three parts in order to increase sorting of small partitions.
Merging Mechanism: Efficiently combines three ordered arrays thus reduces number of comparisons required for sorting.
Performance: It involves in-place merging that is of O(n log n) and little additional memory overhead.

## Randomized Quick Sort

Divide and Conquer Strategy: breaks an array down into smaller pieces and sorts each portion in a recursive manner.
Random pivot selection reduces the chances of the worst-time complexity case scenario.

Performance: The expected worst time complexity for average case scenario is O(n log n) but O(n^2) can be achieved due to the unbalanced partitions.

Performance Comparison

Asymptotic Performance: The asymptotic time complexities model estimates are reasonably close to the performances observed in general.
Constant Factors: The variables that influence actual performance but do not change from one machine architecture or implementation to another.
Coding Complexity and Debugging: This arises because every code has peculiar operations and formations hence it is not possible for every code to be equally complex or have equal debugging challenges.

Remarks

The suitable sorting algorithm depends upon the nature of the dataset or specifications. The speed advantages of Quad-Heap and the optimization of unnecessary space used by Three-Way Merge Sort are evident because of its unique pile arrangement. Despite this potential for worst-case outcomes, Randomized Quick Sort usually yields good average-case results due to its use of random pivots approach. However, in reality, there should consider factors such as desired time complexity, amount of memory required and characterization of datasets. Despite this fact, these algorithms provide a versatile set of ordering options, each option having its own pros and cons.

# Bibliography


- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms (4th Edition)*. Addison-Wesley Professional.
- DAVID K. KAHANER (1979), Fortran Implementation of Heap Programs for Efficient Table Maintenance [Z], https://dl.acm.org/doi/pdf/10.1145/355900.355918
- Randomized Algorithm, Quicksort, https://algoparc.ics.hawaii.edu/~nodari/teaching/f15/Notes/Topic-05B.html
- Probabilistic Analysis and Randomized Quicksort, https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0123.pdf
- Karol Rossa (2022) , GIT 3-way-merge, https://medium.com/@karol.rossa/git-3-way-merge-addc2728c300#:~:text=For%20a%203%2Dway%20marge,goes%20to%20the%20merged%20file
- Alexis Määttä Vinkler, (2023), The Magic of 3-Way Merge, https://blog.git-init.com/the-magic-of-3-way-merge/
- Said Sryheni (2023), Understanding the Randomized Quicksort, https://www.baeldung.com/cs/randomized-quicksort
- Harsil Patel (2022), An Overview of QuickSort Algorithm, https://towardsdatascience.com/an-overview-of-quicksort-algorithm-b9144e314a72
- Hannah C. Tang (2020),Quadtrees, https://courses.cs.washington.edu/courses/cse373/20wi/lectures/11-multidimensional/11-multidimensional.pdf
- https://iq.opengenus.org/quadtree/

## Contributions of Each Team Members:

Aayush Kafle: - Implemented the quick sort and performed the benchmarks for the analysis.

Aashutosh Dahal: - Compiled final report.

Roshan Subedi: - implemented 3-way merge sort.

Joel Gloetzner: - implemented the quad-heap and create test cases.