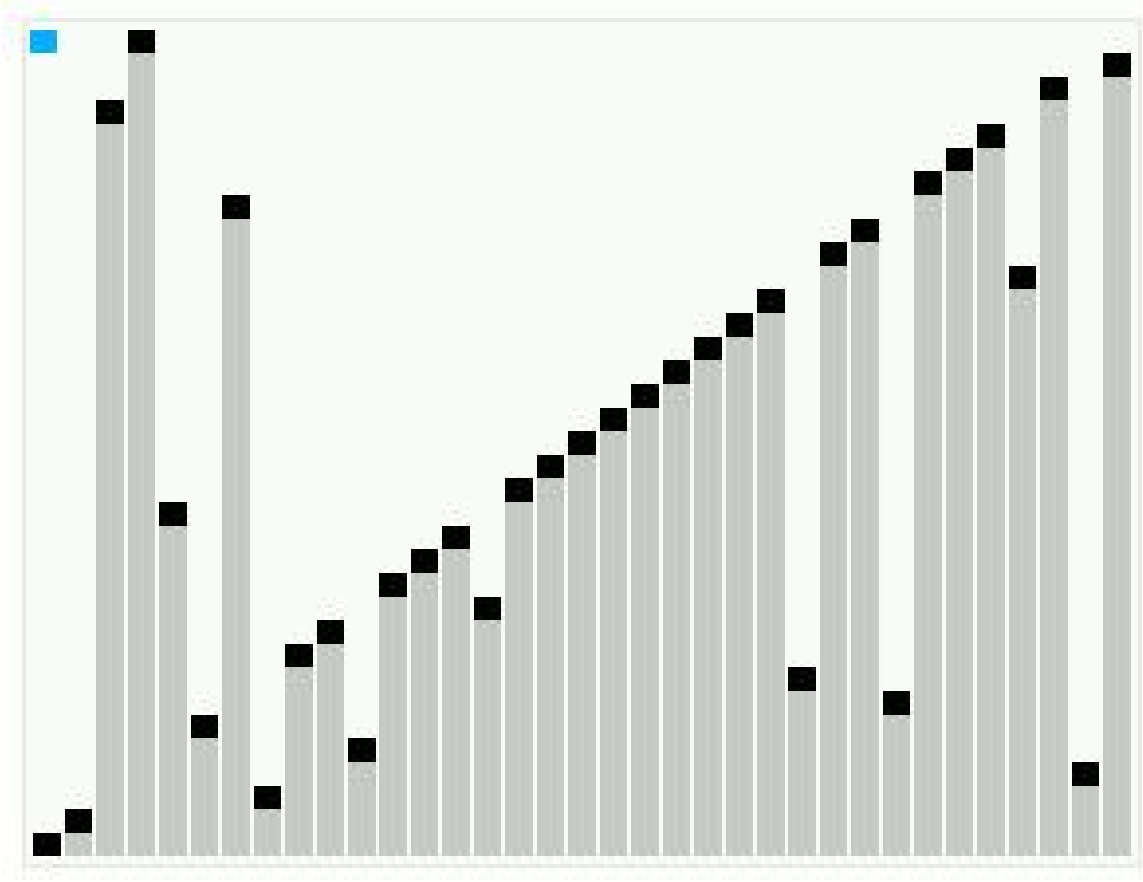
# [Sorting,Analysis]

*By Tejas Menon, Dante Kiaunis and Thomas Honnell*

A multi-faceted analysis of three of the fastest, well-known sorting algorithms in use today.



# Introduction

Sorting has historical underpinnings in the way we use and think about algorithms today. Ever since the shellsort routine was invented in 1959 by Donald Shell[1], the legendary problem has invited the best minds, and produced ingenious solutions still in wide use today. This report will seek to unravel the mysteries of three of these famed $\boldsymbol{\Omega}(nlogn)$ algorithms– the quicksort, mergesort and heapsort– running them on various dataset types to figure their differences when isolated to very specific conditions. This will allow for a versatile understanding of the sorting routines– a programmer would be able to make a decision of which version to use for data specific to their use case which as of currently, is limited to data types fairly distant from common situations. We would like to fill in the gaps for which these three sorting algorithms have no popularly understood benchmark.

The report will move through few phases– first establishing a formal analysis of the predicted performance in running time and space for various data input nuances, then moving to the implementation of the algorithms and testbench, and finally a presentation of the derived results from running the testbench and analysis of the data presented. Throughout too, there will be mentions of implementation differences from theory, and reasons for not pursuing the innumerable improvements that have at the current time, graced each algorithm. This report will thereby analyze the most common version of each of these algorithms– not their elementary conceptions, and also not versions tweaked to the limit. This will allow for a fair reflection of each of the three algorithms in popular use.

# Formal Analysis

Predicting the results of testing will afford us the opportunity to make better decisions regarding design choices prior to implementation. Additionally, we can then decide to make certain reasonable improvements should our analysis predict faster results for a wider range of input data. Predictably also, setting a stage for how a certain algorithm is implemented will provide us the necessary intuition to analyze their performance later on– harking back on these decisions will be helpful in understanding how and why an algorithm fared better/worse than its theoretical calculation. The next subsection continues with a best, worst and average case analysis for the three sorting algorithms, a maximum space requirement analysis, viable improvements to be made and few worked examples to visualize how the sorting would work.

[1] Sorting_algorithm, Shellsort. https://en.wikipedia.org/wiki/Sorting_algorithm

# Heapsort

Heapsort is a comparison-based sorting algorithm invented by J.W.J Williams in 1964[2]. It utilizes a heap data structure to determine a preliminary order, and uses its vaguely sorted structure to extract elements one by one. Most commonly, the heap is represented as an unordered array with a certain array index calculation used to indicate the left and right children of an inner tree node. There are two basic operations in heapsort: the first a preprocessing task that generates a max heap from the input array and second an extraction step that continually removes the root from this max heap before heapifying the tree until the size of the heap becomes zero. Heapsort out of the three sorting algorithms is also unstable due to heap operations disregarding the ordering between two equal elements. For example, take the input array $[6,4,5,1^1,1^2]$ and $[5,4,3,1^1,1^2]$ (already max heaps). Upon calling heapify once, they become $[5,4,1^2,1^1]$ and $[4,1^2,3,1^1]$ respectively. Called once more, they become $[4,1^1,1^2]$ and $[3,1^2,1^1]$. Here is a fundamental problem– which child of the root do we choose to be the first duplicate upon calling max heap again? Since both conditions can manifest totally dependent on the other numbers, it would be impossible or extremely inefficient to keep track of each pair (or more) of duplicates with each heapify call.

The complete sort process can be better understood with a best, average and worst case example:

**Best case:**

In the best case for BuildHeap, the input array contain all duplicates; this means that an array numbered from 0 to size-1 and following the rule that the left child of any internal node is placed at index $2i + 1$ and right child at $2i + 2$ will perform key comparisons exactly $2 * \lfloor size/2 \rfloor$ times i.e. once for the left child and once for the right.

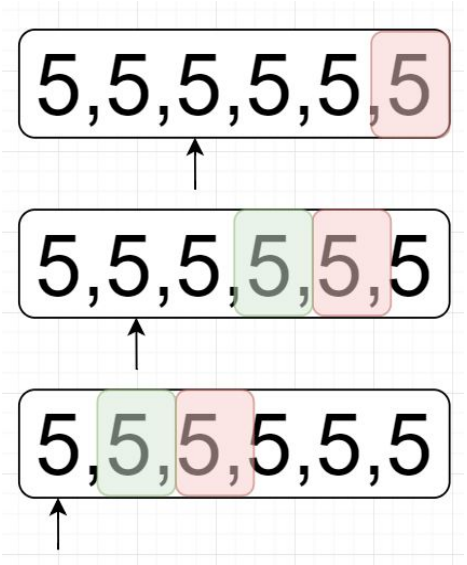



Fig 1: Duplicate build heap procedure: (Arrow points to root. Green for left node and red for right)

Next, heapify would compare each node with its children to see if it is larger than or equal to it. If smaller, it would replace the parent with the largest of its children and call the procedure again on this value. In the best case, after replacing the element at index 0 with one at index size-1, heapify in every case only compares twice to ascertain the correct position of the new root in the array. Therefore, heapify performs $2 * size$ key comparisons to replace all elements at the root. In total then, the

[2] Heapsort. https://en.wikipedia.org/wiki/Heapsort

total amount of comparisons performed is approximately $4 * size \in \Theta(size)$.

**Average Case:**

In the average case for buildheap, there is more work determining to which level a particular heapify call will move the root to. Since the tree is bottom heavy, specifically each level has twice the number of nodes as the previous, the probability of the root settling at level $i$ for a tree of height $H$ and values distributed randomly is thus:

$$P(i, H) = \frac{2^{i-1}}{2^H - 1}$$

The height of the tree here is the number of nodes in the path from the root to the leaf. The level is also assumed to be all values from one to height H.

So, if $2j$ comparisons are made for replacement of an inner root node with one at level $j$ in an inner subtree which is then repeated $2^{i-1}$ times for a particular level on the outer tree, a summation can be set up describing the total number of operations calling the heapify operation on a tree of height $H$:

$$Avg(H) = \sum_{i=1}^{H-1} \Big( \sum_{j=1}^{H-i+1} 2jP(j,i) \Big) 2^{i-1}$$

$$= \sum_{i=1}^{H-1} \Big( \sum_{j=1}^{H-i+1} \frac{j2^j}{2^i - 1} \Big) 2^{i-1}$$

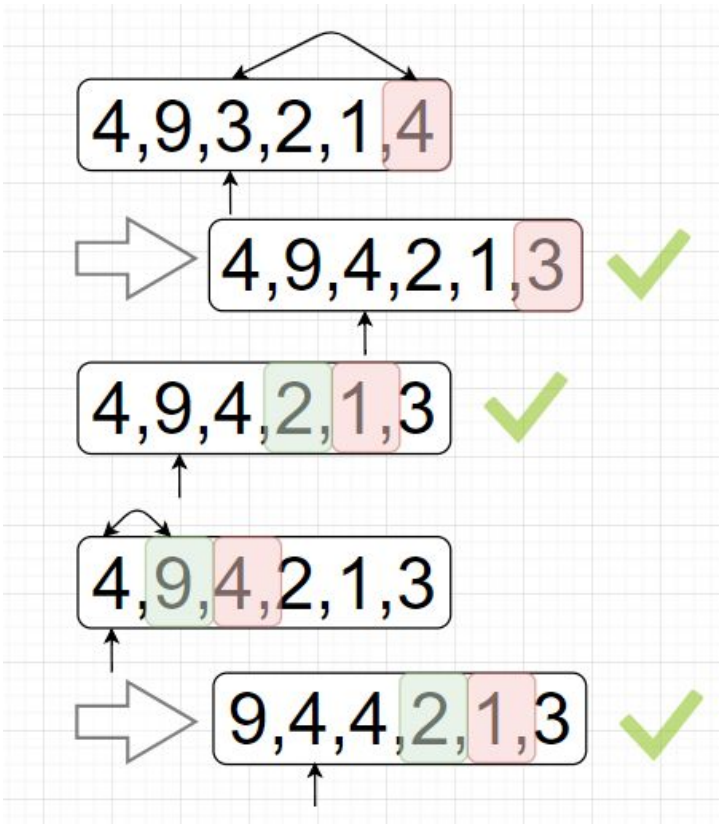
Here it is important to remark about the bounds used for the above equation. The inner summation is executed for as many levels there are in the outer tree minus one: heapify need not be called for all leaves as they are already heapified. Also, $H - i + 1$ is used as the upper bound for the inner summation as the height of the inner subtree is inversely related to the current level of the outer tree i.e. if $i = 2$ and $H = 5$, then the height of my inner subtree is 4 for the $2^{nd}$ level. Solving the above summation substituting $\log_2(N)$ for height yields the following for the average case of buildheap:

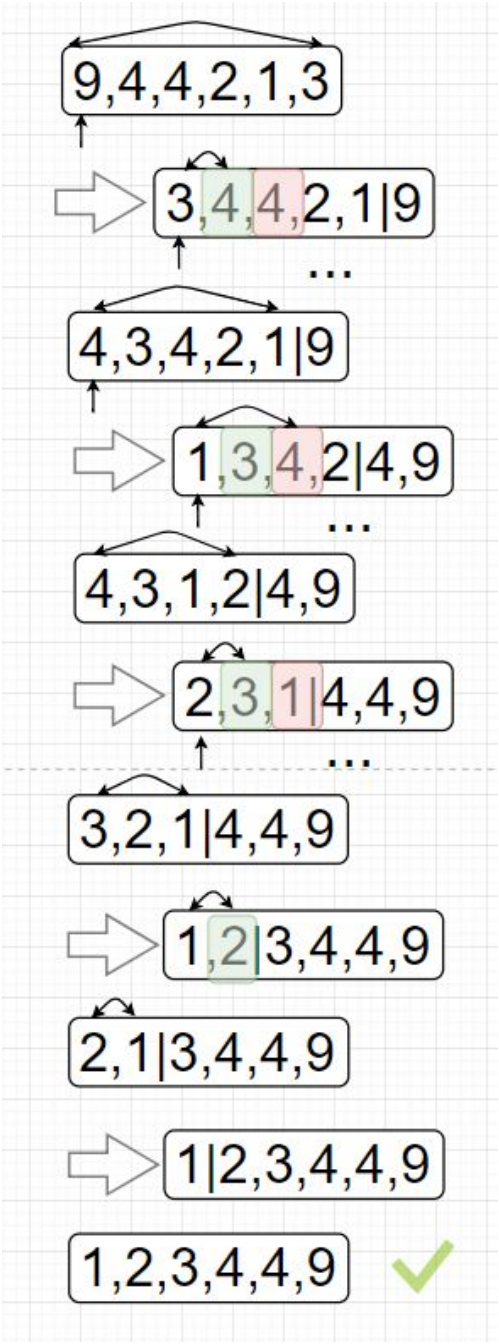
$$Avg(H) = 2H + \frac{2}{H} + 2^{H+1} = 2\log_2(N) + \frac{2}{\log_2(N)} + 2N = \Theta(N)$$

Similarly, we can also prove the average case for heapifying the root $N - 1$ times although this analysis is omitted since it reflects a similar process to BuildHeap. Every time a root node is replaced with the last element, it moves in most cases to the last or second last level thereby incurring less that $H = log_2(N)$ comparisons. Since this process repeats $N - 1$ times, it is reasonable to conclude that heapsort on

average is $\theta(Nlog_2(N))$. Presented next is a quick example to see the average case in action:

Fig 2: Buildheap on the left recursively compares internal nodes to pick the largest among its children and replaces it with the parent. This process continues until the max heap condition is satisfied. On the right, we can see the process of heapsort: the first element is exchanged with the last and the root heapified. For brevity, the entire heapify procedure is not displayed for each root exchange.




**Worst case:**

In the worst case for BuildHeap, we can imagine the input array being in a sorted or somewhat sorted order. In this case, calling heapify on the first $\lfloor size/2 \rfloor$ elements will cause each element to trickle down

the tree to a leaf node in every case therefore making $2 * (H - i)$ comparisons for each node at level $i$. Therefore, the worst case would entail $O(2N) = O(N)$ comparisons as shown by the summation below:

$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$$

Subsequently, heapifying the tree after removal of the maximum element for a total of $N - 1$ times would also require in the worst case $2\lfloor \log_2(n-i) \rfloor$ comparisons each time where $i$ is the number of nodes already extracted and replaced with the last node in the heap. The series of inequalities to get this worst case estimation is outlined below:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2 \sum_{i=1}^{n-1} \log_2 i$$

$$\le 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n \log_2 n.$$

Therefore, heapsort performs in the worst case better than $O(2N \log_2(N) = O(N \log(N))$.

**Space requirement:**

Heapsort is implemented in-place– as shown in Fig 2, every root node extracted can be replaced by the last node in the heap and the size of the heap reduced by one. This results in an algorithm generating an sorted array from the rear end– first the largest element is moved, then the second largest, and so on. In terms of extra space, this means that heapsort can be implemented in $O(1)$, with the cost that the algorithm isn't stable. During implementation however, if in case tail recursion or the loop version of heapsort isn't used to heapify, this could result in potentially an extra space requirement of $O(\log_2(N))$ due to replacement occurring only at a leaf and $H$ stack frames for heapify generated due to each invocation.

**Potential Improvements:**

The algorithm buildHeap in this report implements Floyd's[3] construction of the heap which only heapifies

[3] http://www.cs.princeton.edu/courses/archive/spr13/cos226/lectures/24PriorityQueues-JH-2x2.pdf

internal nodes as the leaf nodes are technically heapified. This reduces by a factor of 2 the total number of loop iterations to build the heap. Similarly for largest node replacement, a bottom-up[4] version of heapsort reduces total internal node comparisons by realizing that replacing the last node from the heap with the root would likely result in one of the smallest elements in the tree moving to a position designated for the largest element which in turn would require traversing a wastefully large number of levels to bring this likely smaller element to the correct position in the tree. Instead, bottom-up heapsort finds the path from the root to a leaf that meets the following condition: every ancestor of this leaf node is larger than or equal to its siblings. This would then allow all insertions to take a lesser amount of comparisons as instead of *replacing* the largest element with the last, we can sift upwards the last element in tree from this particular leaf until it gets to the correct position. In ordinary heapsort as well, this is the final resting place of the last element when sifted top-down. All in all, the bottom-up improvement for heapsort is significant only when comparisons are especially expensive because in every case, the total number of operations conducted during a heapify call is the same (bottom up heapsort trades the number of comparisons required with assignment operations instead).

**Implementation:**

We decided for our version of heapsort to use Floyd's buildheap improvement as it provided for a significantly improved constant factor but decided not to implement the bottom up heapsort as we were only comparing integer data. Listed below is code in Java for our heapsort procedure:

```
public class Heapsort {
    public static int[] heapsort(int [] input) {

        if (input == null) return null;

        int size = input.length;
        for(int i = (size/2)-1;i>=0;--i)  {
            heapify(input,i, end: size-1);
        }
        while(size>1) {
            int temp = input[0];
            input[0] = input[size-1];
            input[size-1] = temp;
            --size;

            heapify(input, start: 0, end: size-1);
        }
```

Fig 3: Slightly unconventional, this algorithm calls heapify to build the heap after which a while loop replaces the root node to the end of the heap width.

[4] Bottom-up Heapsort. J. W. J. Williams: *Algorithm 232 – Heapsort*. In: *Communications of the ACM*, 1964, 7(6), S. 347–348.

```java
public static void heapify(int []input, int start, int end) {

    int left = start*2 + 1;
    int right = start*2 + 2;

    boolean l = true;
    boolean r = true;

    if (left > end) l = false;
    if (right > end) r = false;

    if (!r && !l) return;
    if (r && l) {
        if (input[right] > input[left]) l = false;
        else r = false;
    }
    if (r && input[right] > input[start]) {
        int temp = input[right];
        input[right] = input[start];
        input[start] = temp;
        heapify(input, start: 2*start + 2,end);
    }
    if (l && input[left] > input[start]) {
        int temp = input[left];
        input[left] = input[start];
        input[start] = temp;
        heapify(input, start: 2*start + 1,end);
    }
}
```

Fig 4: Heapify implementation in Java. Note that alternative implementations can make use of tail recursion to avoid stack frame accumulation as would be in this case (Every time a right node is traversed to, a frame adds to the stack).

# Testing

Efficiently testing these three algorithms required a great deal of planning and plotting– how to comprehensively check for performance efficiency under all use cases a sorting algorithm may be subject to? How to ensure minimal timing latency for a certain data size and type? And whilst optimizing for these factors, how to remain under time and memory constraints that are eventually limited?

During research, we discovered the industry-wide usage of OpenJDK JMH[5] for reliable, accurate timing analysis in Java– testbenches that warm up the cache and run trial iterations multiple times before executing the function to benchmark. Additionally, we noticed its isolated thread performance which meant that processes running on other threads wouldn't affect its performance as long as they were

[5] https://openjdk.java.net/projects/code-tools/jmh/

limited to a minimum. The only problem that accompanied these highly functional, complex tools was its learning curve–tying its features into our specific requirement would require mighty tinkering and the willingness to be able to take the risk of planning our testbench later–we would only be able to use any of its functional advantages after carefully understanding the API. Due to the shortage of time, we decided to forgo this study and rather innovated to bring these advantages to our own homegrown test bench. The specific advantages we developed for was minimal latency and maximum coverage whilst reducing usage of resources as much as possible. Code examinations below are for notable snippets from our testbenches Testdata and Testrunner; Testdata generates integer arrays of size $1.5^N$ of 12 different data types (storing them for later use by another algorithm) and Testrunner actually performs the execution measurement and writes execution timings into an external file:

**Request Function from Testdata (See Appendix)**

First, it's important to understand the data structure: three vector arrays nested within one another allow for multilevel retrieval. The first level stores arrays of the 12 data types (definitions of each type are expressed in the header of the file), the second level stores arrays of increasing sizes (array index i stores integer arrays of size $1.5^i$) and the third level stores different integer arrays of the same size. Ostensibly, these types would encompass most variations of data in the real world and storing these arrays mean that we can use the *same* data to benchmark all our algorithms. Although a seemingly promising plan, analysis of the run function in Testrunner will describe certain problems with this strategy and why we eventually resorted to performing our analysis singularly–not maintaining storage and deallocating memory Testdata acquires after every execution.

Now we can discuss the specifics of function Request. First, a lookup integer is assigned to the corresponding index of alldata that data of type ‘type’ belongs in. Then, a type reference holds the Vector of Vectors i.e. all data of a certain type. Looking up at this index, we always know that it exists as the Vector is minimally this size. Next, we move to a loop sequence that guarantees procurement of data– for as long as data at index base1size doesn’t exist in typeref (i.e. typeref is a smaller array) we keep adding empty Vectors to typeref until a sizeref at index base1size exists. Next an identical loop proceeds to fill sizeref until we have an integer array present at index indexforsize. Note that all sizeref indices $\leq$ indexforsize have integer arrays of the same length: $1.5^{base1size}$. The usage of these exceptions are so that in case data does exist at a certain index (it has been previously requested before) we can quickly serve it without have to call create().

**Run and exec functions from Testrunner (See Appendix)**

The way we intended for our testbench to run is as follows: for the three algorithms and 12 input data types, run five different arrays for each size going up to size too large to be allocated or a size taking $> 10$ seconds to produce data or a size taking $> 10$ seconds to perform the sort in which case we/the system would raise an exception and switch to the next data type. We were shortsighted however, on two

counts– storing multiple integer arrays where $i > 45$ i.e. $1.5^{45}$ integers $\sim 336$ MB results in quick accumulation of large sums of data which raises the java.noMemoryError forcing program termination. This meant that storing arrays for future algorithms to use was a big no unless we were willing to limit max array size to a modest figure and therefore reduce our 'coverage'. Upon inspection, we realized that foregoing the ability to run the same data on the three different algorithms would be a better option than foregoing data points for $i > 45$ since we felt it was only for data of this size that would display tightness to the $n\log(n)$ line and smaller values may produce unreliable results due to shorter running times being prone to measurement inconsistencies. For the future, we hoped to write these arrays to disk rather than storing them on memory, in which case the out of memory error would not be a problem.

Therefore, for our revised version of run() we created and released an instance of Testrunner for each array we measured– since the randomness of the data was seeded by System.nanotime(), we wouldn't have to worry about duplicate data generation and since the memory was released, arrays of size $1.5^{50}$ integers $\sim 2.5$ GB was able to be generated for data types that didn't trigger the 'Large Time' exceptions caused by $> 10$ second latency producing or sorting data. Important to clear up here also is the reason for evaluating multiple integer arrays of the same size– no one array is a perfect sample for a particular data type and therefore taking the average of the minimum time execution for each array would give us the most reliable benchmark.

Finally, the exec() function called by run() is also highly critical to the function of the program. Initially, to ensure execution time for a particular array size was stable (and therefore minimal), plans were to keep measuring execution until ten successive measures showed up the same value after which minimal time execution would be chosen from a further 0.5 seconds of measuring. These plans were foiled as we realized the instability of cpu threads– it was highly improbable that even two successive execution measurements would return the same value for input sizes that were larger than trivial. Due to these reasons, we resorted to measuring minimal execution time from repeated executions lasting 0.5 seconds. If an execution lasted 10 seconds or more, an exception was thrown. Also, if in case an execution lasted 0 seconds (reported by System.nanotime()), we would reset the timer and rather than measuring minimum time execution from a further 0.5 seconds, a counter would be started that measures the number of executions for the next 0.5 seconds which is then used to calculate the average time taken per execution.

To confirm our algorithms were implemented correctly, we also used a confirm() function that once the execution time for an array was measured, checked that data at each index was either equal or larger than the index before it. If not, it would throw an Unsorted exception which would be caught by the outermost catch block in run() and the program terminated.

# Analysis

From conducting a standard test run lasting about four hours, we obtained 36 .csv files– 1 for each algorithm running on each data type. We decided to graph them using scatter plots on the primary y-axis grouping each graph by data type. On the secondary axis, we graphed $N\log(N)$. This allowed us to make a rudimentary guess as to whether an algorithm was behaving suspiciously at all. Each of the obtained graphs should be available in the appendix. This section will refer to the particular graph it draws its analysis from.

All in all, results weren't unexpected although we could ascertain some clear winners for the majority of data types (at least as per the test bench and algorithm implemented by this project). With the exception of totally duplicate data, Heapsort performed the slowest– about 2.5x slower than quicksort and 1.5x slower than mergesort. Although there weren't clear indicators of such a large performance difference since we didn't look at the exact average case time complexity for quicksort or mergesort, the improvements made to quicksort in our implementation– Hoare's partition scheme and randomized pivot –has meant that the worst case of $O(n^2)$ is much harder to come by whilst ensuring array partitions are of approximately equal size. Additionally, due to inherent locality advantages of quicksort such as accessing array elements next to one another much more often than those further away (cache calls) means that quicksort's cost per operation is significantly lesser than that of heapsort which accesses array indexes usually much further away. In addition, quicksort's in-place performance means that allocation time of an output array and assignment operations to further away memory locations is not an issue as is the case with mergesort, which can more or less distinguish their performance differences. The differences between heapsort and mergesort can again be figured by memory locale– mergesort comparatively accesses array indices closer to one another (merging only looks at successive indices for example) while heapsort's left and right children on large arrays may be several memory blocks ahead. Another caveat that could be magnifying the shortcomings of this implementation of heapsort is recursive heapify– for each function invocation, new stack memory and addressing needs to be allocated that along with space, costs additional time.

Explaining however, the stellar performance of heapsort when exposed to totally duplicate data (one data value) is simple: buildheap and recursive heapify ran in $\theta(N)$ since a parent node never exchanges with any of its children if they are all the same value. Another Interesting relationship is the increasing variance between all three algorithms with data that is random and unique (more the unique and random, more the variance). The reverse also seems to be true– data that is sorted and duplicated have lesser variance. More than a finding that differentiates between the three algorithms, this seems to suggest the nature of extreme data in bringing together similar performing algorithms.

When each graph is viewed individually among all their various options for data type, a data type hierarchy is also visible– random data sorts slowly for mergesort and heapsort which then slowly improves with an almost sorted structure and totally sorted, duplicate data sorts fastest on these algorithms. With mergesort, it is clear why this is the case–sorted data is its best case input and random, unique data producing increased chances for its worst case, with almost sorted data performing as expected, somewhere in the middle. Mergesort also shows the highest concentration of lines thereby describing its consistency– it performs at $\theta(N \log(N))$ in every case. With heapsort, we see that it has most variance between its lowest and highest data values and lines equally spread across the range. Its improved performance on sorted input can be explained for the reverse case: buildheap occurs in $\theta(N)$ due to the non-presence of nodes smaller than their children. For quicksort however, a large quantity of the lines are concentrated at 1000 ms and then distributed upwards more sparingly. This explains its overall fastness when compared to the others.

# Conclusion

Quicksort being called the gold standard for good sorts can thus be regarded as not an incorrect statement. In addition to being the fastest and in-place, it can also be made stable if needed (at the loss of in-placeness). Admittedly this project did not use a version of heapsort completely optimized for comparison, but as regarded by other researchers, quicksort when implemented correctly presents an optimization 2-3 times faster than heapsort. In multiple domains however, heapsort can continue to have a lasting impact– sorting data contained in sparse, or binary data sets with many duplicate values; usage of heapsort for data already represented or available as a priority queue for example in the best-first-search algorithm to find routes or for bandwidth allocation in computer networks. Mergesort as discovered seems to be an algorithm highly consistent and to a high degree independent to data input. It would be recommended by this project as an algorithm to be used for security, or in other situations where the algorithms performance should not be adversely affected by data input. An example would be flight schedules available to the ATC sorted by distance from the airport. If the availability of this information could be compromised by an adversarial attack on the input, flight detail records could update very slowly. Additionally too, this is not to undermine the usage of elementary sorts such as the insertion sort– for smaller array sizes, they can provide the same benefits as the $N \log(N)$ algorithms in addition to being stable and somewhat faster due to no recursive calls and therefore less space used. Such algorithms predicted usage is thus in embedded, low level applications were sort sizes may not be high and resources limited. Overall, the guiding principle to making these decisions is to analyze the constant factor hidden from Big Oh expressions– if array sizes are small, these values will most likely impact runtime performance.

# Appendix

This appendix contains all testing functions from Testrunner and Testdata followed by all graphs obtained from analysis– 12 broken down by data type and 3 broken down by algorithm.

```
/*Testdata.java class implementation last edited 3/3/19. This class produces
data and stores
 its results for later use. Testrunner only uses one instance of this class
at a time so that
 RAM usage is minimal when allocating very large arrays. Data output
generated is of types:
 totalrand : Totally random integer data => every integer has equal
probability for appearing at any index
 totaldup: Every integer has equal probability for appearing at EVERY index
of this array
 almostsortdup: Array generated with good chance of duplicates in an almost
sorted order
 sortdup: Array generated with good chance of duplicates in sorted order
 revsortdup: Array generated that is reverse sorted with good chance of
duplicates
 almostrevsortdup: Array generated that is almost reverse sorted with good
chance of duplicates
 randup: Random array generated with good chance of duplicates
 almostsortuniq: Array generated that is almost sorted with no duplicates
 sortuniq: Array generated that is sorted with no duplicates
 revsortuniq: Array generated that is reverse sorted with no duplicates
 almostrevsortuniq: Array generated that is almost reverse sorted with no
duplicates
 randuniq: Array generated that as completely random with no duplicates
*/
import java.util.Random;
import java.util.Vector;

public class Testdata {
   private Vector<Vector<Vector<int[]>>>  alldata; //an array of int[] within
an array within an array
                                                   //First level array varies
between the different data types
                                                   //Second level array
varies between different sizes (actual size is
                                                   //1.5^(size). Third level
```

```
array varies between arrays of the same size
                                                        //Using a vector means
that you can construct an extendable array
   static Random rand;

   public Testdata() throws java.lang.RuntimeException,
java.lang.OutOfMemoryError{
       alldata = new Vector<Vector<Vector<int[]>>>(12);
       rand = new Random(System.nanoTime()); //every instance of Testdata
produces different random values
       for (int i = 0;i<12;++i) {
           alldata.add(new Vector<Vector<int[]>>());
       }
   }

   public int[] request(int base1size, int indexforsize, String type) throws
java.lang.RuntimeException,
           java.lang.OutOfMemoryError {

       if (base1size < 0 || indexforsize < 0) throw new RuntimeException();

       int lookup;
       if (type.equals("totalrand")) lookup = 0;
       else if (type.equals("totaldup")) lookup = 1;
       else if (type.equals("almostsortdup")) lookup = 2;
       else if (type.equals("sortdup")) lookup = 3;
       else if (type.equals("revsortdup")) lookup = 4;
       else if (type.equals("almostrevsortdup")) lookup = 5;
       else if (type.equals("randup")) lookup = 6;
       else if (type.equals("almostsortuniq")) lookup = 7;
       else if (type.equals("sortuniq")) lookup = 8;
       else if (type.equals("revsortuniq")) lookup = 9;
       else if (type.equals("almostrevsortuniq")) lookup = 10;
       else if (type.equals("randuniq")) lookup = 11;
       else throw new RuntimeException();

       Vector<Vector<int[]>> typeref = alldata.elementAt(lookup); //lookup an
array of different sizes for a particular data type

       boolean correctsize = false;

       //Keep adding arrays for every size until an element at
typeref.elementAt(base1size) exists
```

```
        while (correctsize == false) {
            try {
                Vector<int[]> sizeref = typeref.elementAt(base1size);
                correctsize = true;

                boolean correctindex = false;

                //Keep adding int[] of a particular size and data type until
you get to an array at sizeref.elementAt(indexforsize)
                while (correctindex == false) {
                    try {
                        int[] requested = sizeref.elementAt(indexforsize);
                        correctindex = true;

                        return requested;
                    } catch(ArrayIndexOutOfBoundsException e) {
                        sizeref.add(create(lookup,base1size));
                    }
                }
            } catch (ArrayIndexOutOfBoundsException e){
                typeref.add(new Vector<int[]>());
            }
        }
        return null;
    }

    //Create an array of type as shown below and for a particular size
    private int [] create(int type,int base1size) throws
java.lang.OutOfMemoryError,RuntimeException {
        if (type == 0) return totalrand(base1size);
        else if (type == 1) return totaldup(base1size);
        else if (type == 2) return almostsortdup(false,base1size);
        else if (type == 3) return sortdup(false, base1size);
        else if (type == 4) return sortdup(true, base1size);
        else if (type == 5) return almostsortdup(true,base1size);
        else if (type == 6) return randup(base1size);
        else if (type == 7) return almostsortuniq(false,base1size);
        else if (type == 8) return sortuniq(false,base1size);
        else if (type == 9) return sortuniq(true,base1size);
        else if (type == 10) return almostsortuniq(true,base1size);
        else if (type == 11) return randuniq(base1size);
        else throw new RuntimeException();
    }
```

```
    private int [] totalrand(int base1size) throws java.lang.OutOfMemoryError{
        int size = (int)Math.pow(1.5,base1size);
        int [] arr = new int[size];

        for(int i = 0;i<size;++i) {
            arr[i] = rand.nextInt();
        }

        return arr;
    }

    private int [] totaldup(int base1size) throws java.lang.OutOfMemoryError{
        int size = (int)Math.pow(1.5,base1size);
        int [] arr = new int[size];

        int randnum = rand.nextInt();
        for(int i = 0;i<size;++i) {
            arr[i] = randnum;
        }

        return arr;
    }

    private int [] rev(int [] arr) throws java.lang.OutOfMemoryError {
        for (int i = 0, j = arr.length-1;i<j;++i,--j) {
          int swap = arr[i];
          arr[i] = arr[j];
          arr[j] = swap;
        }
        return arr;
    }

    private int [] almostsortdup(boolean reverse,int base1size) throws
java.lang.OutOfMemoryError{
        int [] arr = sortdup(reverse,base1size);

        for (int i=0;i<arr.length;++i) {
            int random = rand.nextInt(100) + 1;
            int random2 = rand.nextInt(arr.length);
            if (random < 15) {
                int to_swap = arr[random2];
                arr[random2] = arr[i];
```

```
                arr[i] = to_swap;
            }
        }
        return arr;
    }

    private int [] sortuniq(boolean reverse, int base1size) throws
java.lang.OutOfMemoryError{
        int size = (int)Math.pow(1.5,base1size);
        int [] arr = new int [size];


        long choosefrom = (long)Math.pow(2,32) - (size - 1);
        long last_added = 0;

        for(int i = 0; i<size; ++i) {
            last_added += Math.floorMod(rand.nextLong(), (choosefrom -
last_added)) + 1;
            ++choosefrom;
            int to_int = (int)(last_added - (Math.pow(2,32)-Math.pow(2,31) +
1));
            arr[i] = to_int;
        }

        if (reverse == true) rev(arr);
        return arr;
    }

    private int [] randuniq(int base1size) throws java.lang.OutOfMemoryError{
       int [] arr = sortuniq(false,base1size);

       for(int i = 0;i<arr.length;++i) {
           int swapindex = rand.nextInt(arr.length);
           int swap = arr[swapindex];
           arr[swapindex] = arr[i];
           arr[i] = swap;
       }
       return arr;
    }

    private int [] almostsortuniq(boolean reverse,int base1size) throws
java.lang.OutOfMemoryError{
        int [] arr = sortuniq(reverse,base1size);
```

```
        for (int i=0;i<arr.length;++i) {
            int random = rand.nextInt(100) + 1;
            int random2 = rand.nextInt(arr.length);
            if (random < 15) {
                int to_swap = arr[random2];
                arr[random2] = arr[i];
                arr[i] = to_swap;
            }
        }
        return arr;
    }

    private int [] randup(int base1size) throws java.lang.OutOfMemoryError {

        int [] arr = sortdup(false,base1size);

       for(int i = 0;i<arr.length;++i) {
           int swapindex = rand.nextInt(arr.length);
           int swap = arr[swapindex];
           arr[swapindex] = arr[i];
           arr[i] = swap;
       }
       return arr;
    }

    private int [] sortdup(boolean reverse,int base1size) throws
java.lang.OutOfMemoryError{
        int[] arr = sortuniq(reverse, base1size);

        for (int i = 0; i < (arr.length - 1); ++i)
            if (Math.floorMod(rand.nextInt(), 3) == 1) arr[i + 1] = arr[i];

        return arr;
    }
}
```

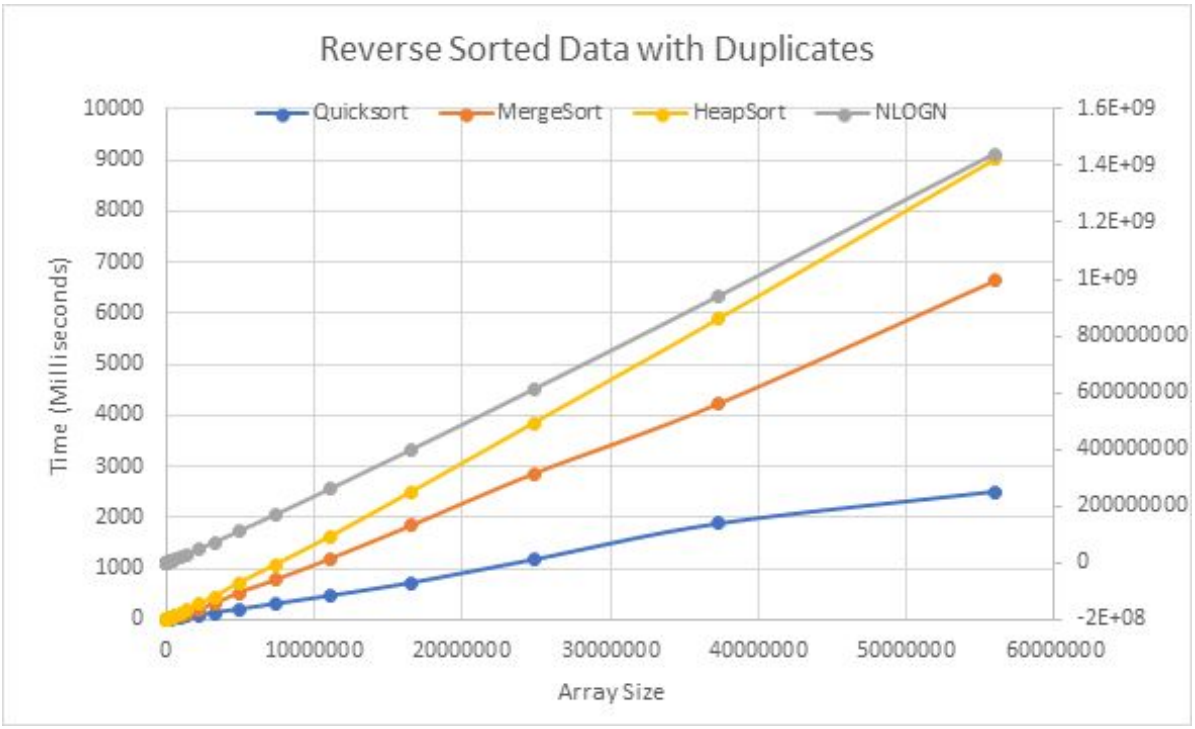
```
/* Testrunner.java class implementation last edited 3/2/19. This file runs
all the tests
   for the three sorting algorithms and outputs .csv files with data points
for input size and
   execution time.
 */

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Testrunner {
    private Testdata data; //data object used to generate psuedo random
arrays in various configurations

    //The main function that runs all testcases for the three algorithms i.e.
5 times for each size
    //for each input data type for each algorithm. There is no set limit on
how large the size for
    //generating arrays can reach provided no exception is hit for the
reasons specified on line 49
    public void run() {
        try { //see Testdata.java class for details om each data type it
serves
            String[] input = {"totalrand", "totaldup", "almostsortdup",
"sortdup", "revsortdup",
                    "almostrevsortdup", "randup", "almostsortuniq",
"sortuniq", "revsortuniq", "almostrevsortuniq", "randuniq"};
            String[]algo = {"MergeSort","HeapSort","QuickSort"};

            for (int k = 1; k < 2; ++k) { //For each of the three algorithms

                for (int i = 0; i < input.length; ++i) { //For all input data
types

                    BufferedWriter bw = new BufferedWriter(new
FileWriter("./dataout/"+algo[k] + "_" + input[i] + ".csv"));
                    boolean exceptionhit = false; //exception hits occur due
to various reasons
                    int size = 0; //floor(1.5^(size)) represents actual
integer array size

                    while (!exceptionhit) { //increments size until an
```

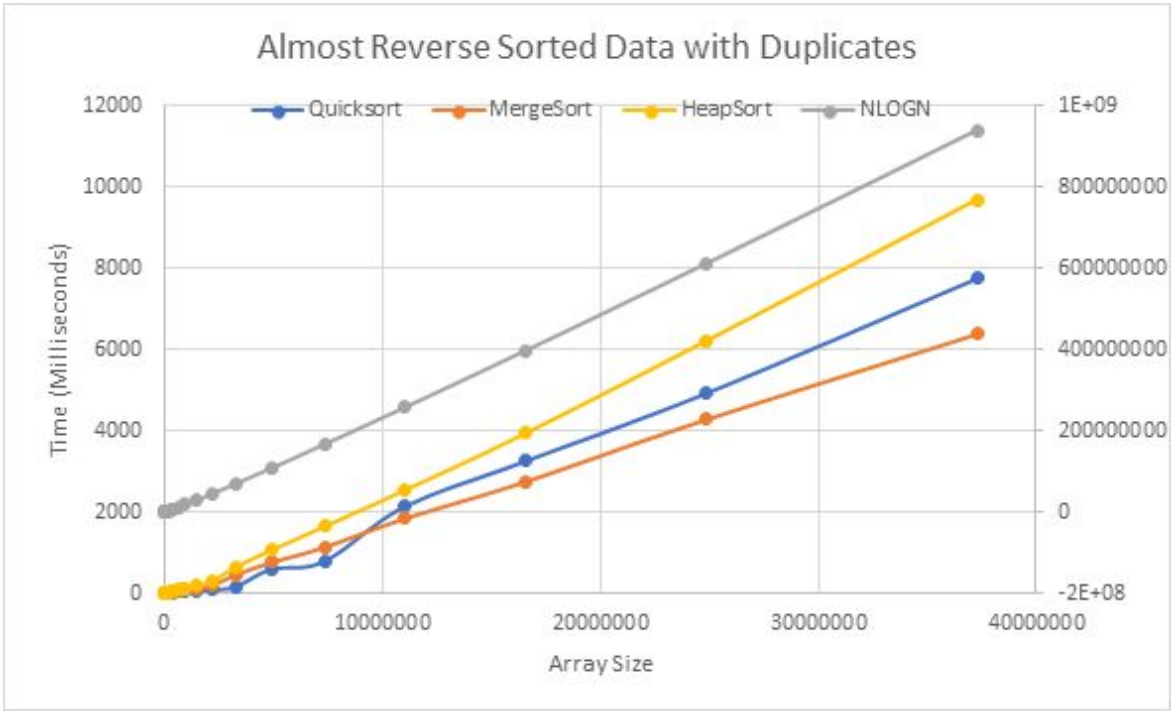
```
exception is hit
                        try {
                            long sum = 0;
                            long avg = 0;
                            for (int j = 0; j < 5; ++j) { //repeat
calculation for 5 arrays of size 'size'
                                Testdata data = new Testdata(); //a new
testdata object for every array minimizes RAM usage
                                long getdata = System.nanoTime(); //benchmark
time taken to allocate array
                                int[] arr = data.request(size, 0, input[i]);
//array of 'size' in returned as per input[i]
                                long gotdata = System.nanoTime();
                                if (gotdata - getdata > 10e9) throw new
RuntimeException(); //5 seconds max to get array
                                sum += exec(arr,k); //exec returns time taken
to sort array
                                System.out.print(algo[k] + "_" + input[i] +
"_" + size + "_" + j + " Done! ");
                            }
                            System.out.println();
                            avg = sum / 5;
                            bw.write(size + "," + avg);
                            bw.newLine();
                            bw.flush();
                            ++size;
                        }
                        catch (RuntimeException ex) { //exception is reached
in three cases: data retrieval > 10 sec
                                                      //Out of memory error on heap
due to an array too large
                                                      // >10 sec per benchmark
                            exceptionhit = true;
                        }
                        catch (Error e) {
                            exceptionhit = true;
                        }
                    }
                    bw.close();
                }
            }
        } catch(IOException ex) {
            System.out.println(ex.getMessage());
```

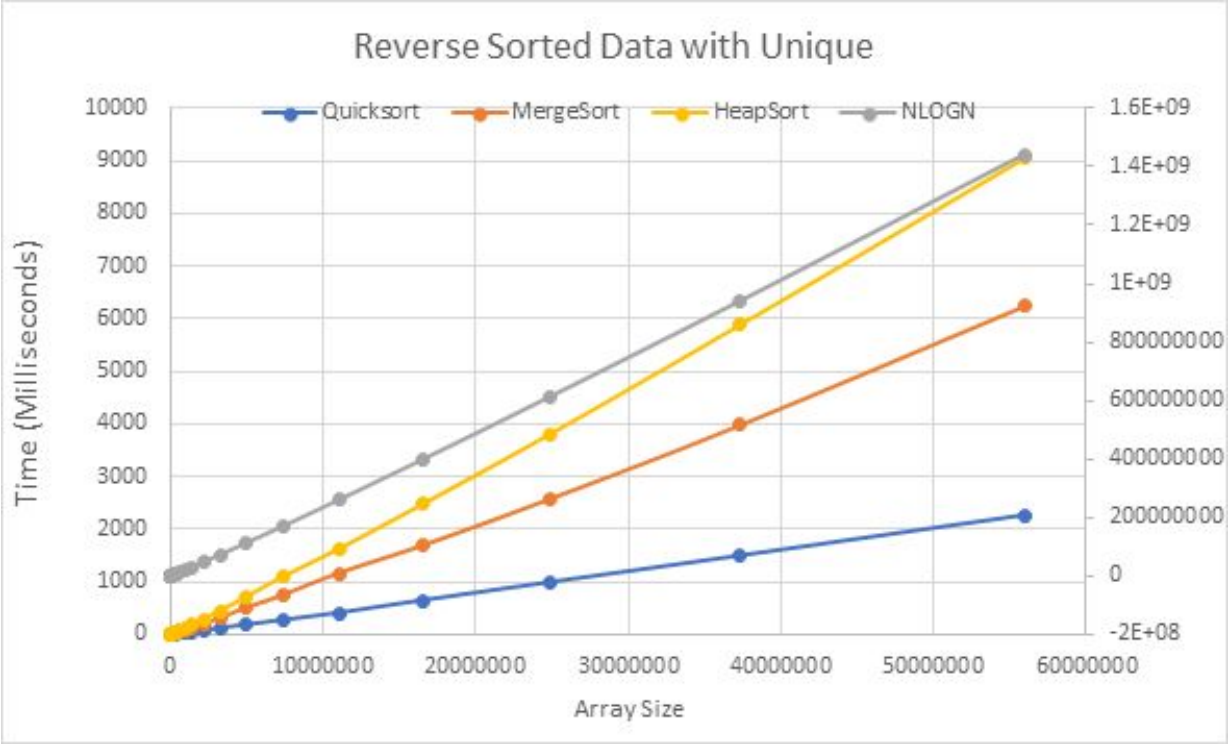
```
        }
        catch(UnsortedException ex) {
            System.out.println(ex.getMessage());
        }
    }

    //This function runs a particular input for a particular algorithm for
0.5 seconds and
    //returns the minimum execution time from those runs. If any one
execution lasts more than
    //10 seconds the execution is aborted and exception raised. Similarly, if
any one execution
    //reports 0 nanoseconds of time taken, a counter is started for all
future executions and executions
    //measured for a further 0.5 seconds before returning the average time
taken per '0' second execution
    public long exec(int [] input, int choice) throws RuntimeException,
UnsortedException{
        double alldiff,counter,thisdiff,loopstart,mindiff;
        loopstart = counter = alldiff = thisdiff = 0;
        mindiff = 10e9;
        int [] sorted_arr = null;
        double thistart, thisend;
        boolean waszero = false;

        loopstart = System.nanoTime();
        while(alldiff < 5e8) {
           thistart = System.nanoTime();
           switch (choice) {
               case 0:
                   sorted_arr = Mergesort.mergesort(input);//mergesort
                   break;
               case 1:
                   sorted_arr = Heapsort.heapsort(input); //heapsort
                   break;
               case 2:
                   sorted_arr = Quicksort.quicksort(input) ; //quicksort
                   break;
           }
           thisend = System.nanoTime();
           thisdiff = thisend - thistart;
          // System.out.println(thisdiff);
```

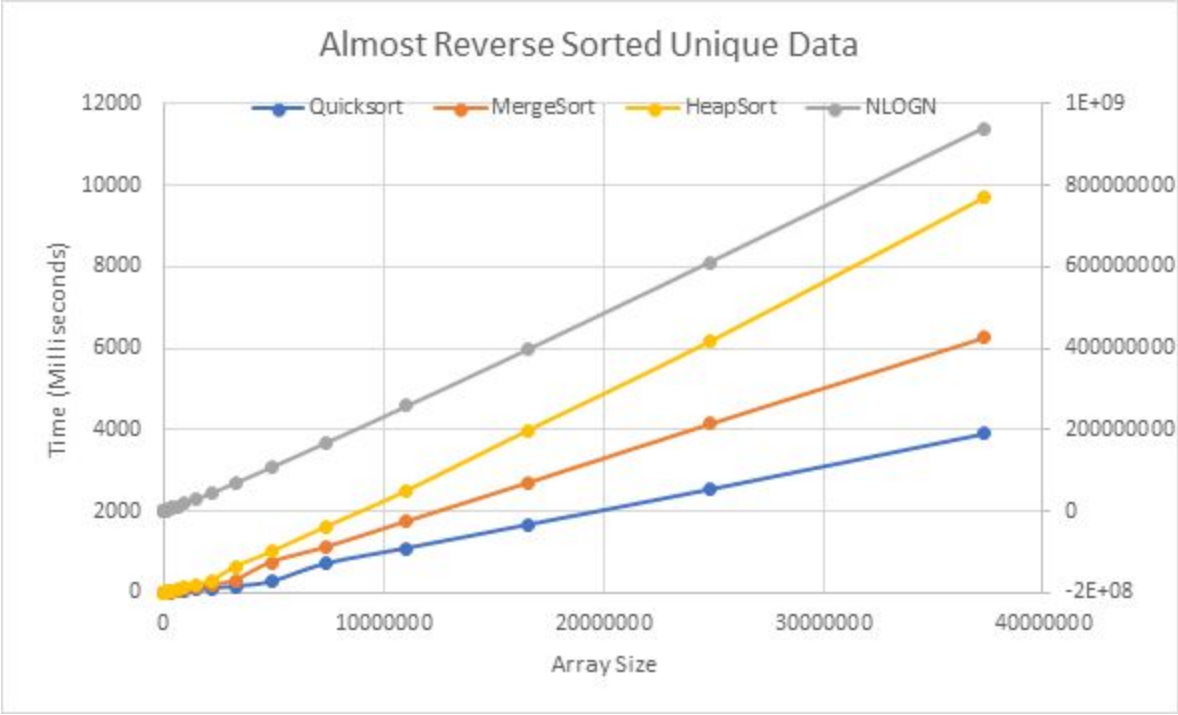
```
            if (thisdiff > 10e9) {
                System.out.println("LARGE TIME");
                throw new RuntimeException();
            }

            if(thisdiff == 0 && waszero == false) { //everything reset for the
first '0' nanosecond execution
                counter = 1;
                waszero = true;
                loopstart = System.nanoTime(); //0.5 seconds of execution
measured from here onwards
            } else if (waszero)
                ++counter;
            else {
                if (thisdiff < mindiff) mindiff = thisdiff;
            }
            alldiff = thisend-loopstart; //alldiff measures time taken since
loop execution or first zero second measurement
        }

        if (!confirm(sorted_arr)) //if unsorted array is detected due to
faulty algorithm
            throw new UnsortedException();

        if(waszero) return (long) (alldiff/counter); //returns the average of
all runs in case we encounter a 0 s execution
        else return (long) mindiff;
    }

    //placeholder function to be replaced b sorting functions
    /*public int[] placeholder(int [] input) {

        int val = (int)(Math.log(input.length)/Math.log(1.5));

        for(int i = 0;i<input.length;++i) {
            input[i] = i;
        }

        return input;
    }*/

    //checks if output is in sorted order
    public boolean confirm(int [] to_check) {
```

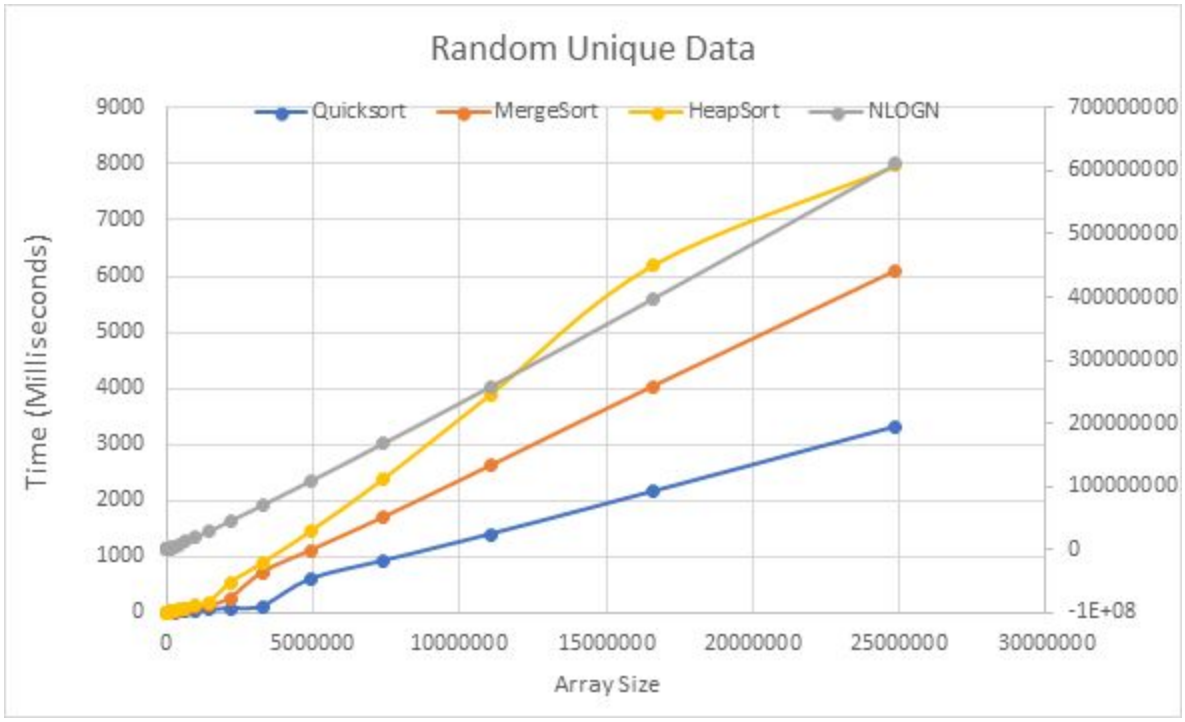
```
        if (to_check.length == 1) return true;
        for (int i = 1; i < to_check.length; ++i)
            if (to_check[i] < to_check[i - 1]) return false;

        return true;
    }
}
```

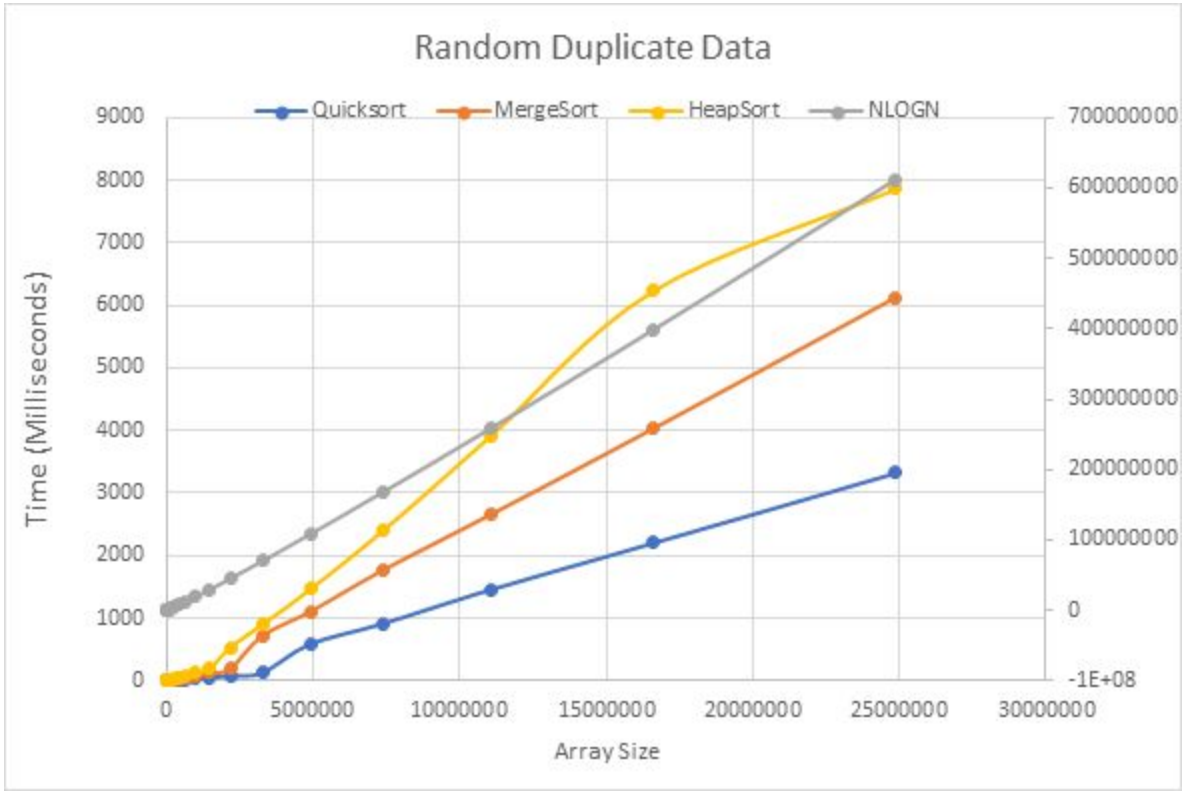

Reverse Sorted Data with Duplicates
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1.6E+09
1.4E+09
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
-2E+08
0
10000000
20000000
30000000
40000000
50000000
60000000

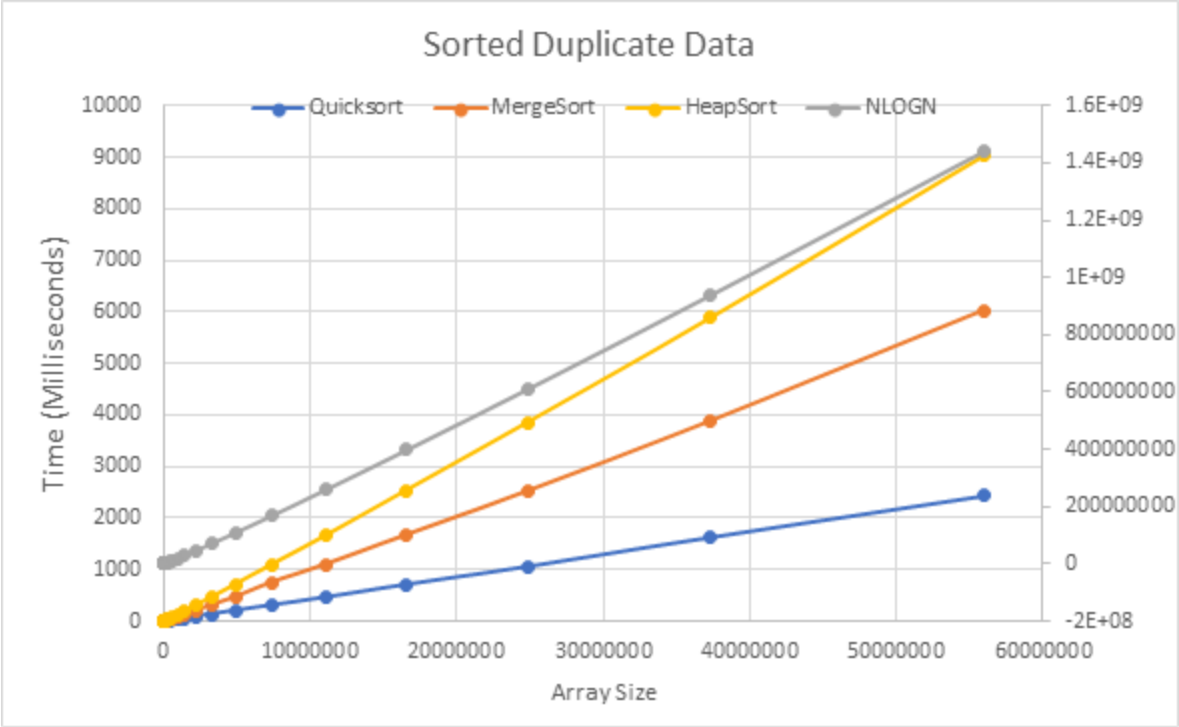

Almost Reverse Sorted Data with Duplicates
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
12000
10000
8000
6000
4000
2000
0
1E+09
800000000
600000000
400000000
200000000
0
-2E+08
0
10000000
20000000
30000000
40000000

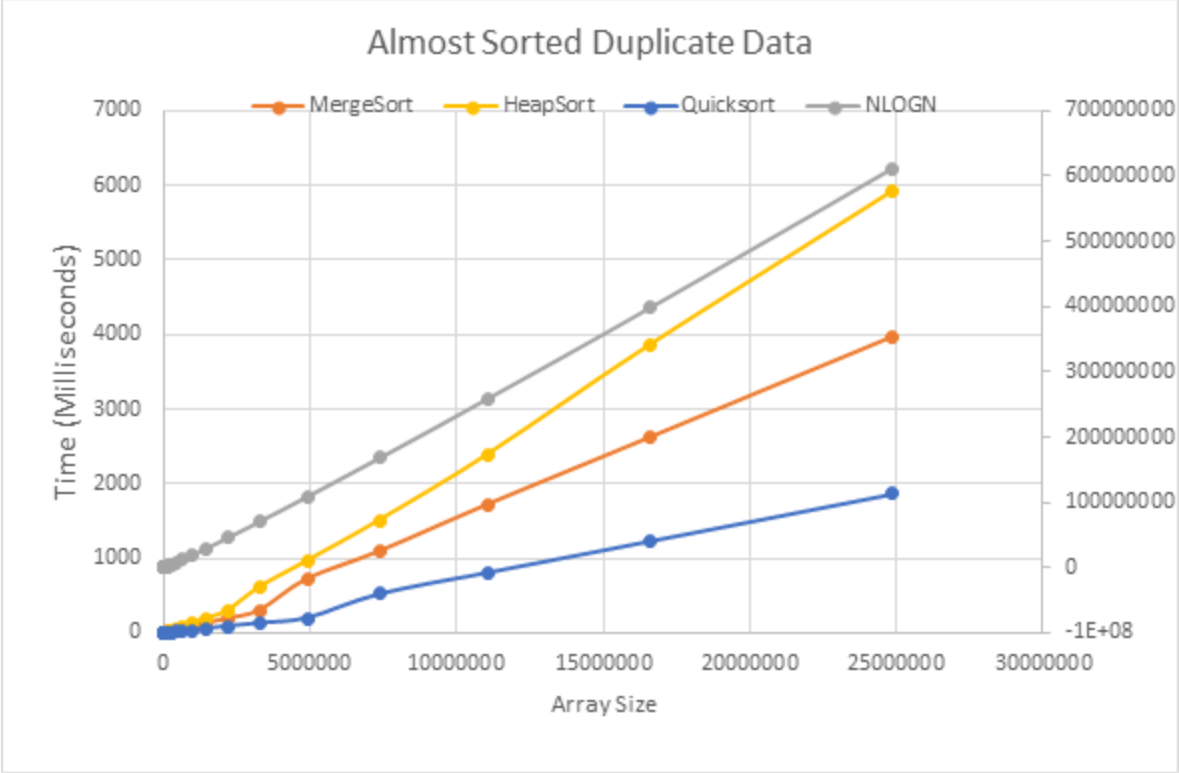

Reverse Sorted Data with Unique
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1.6E+09
1.4E+09
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
-2E+08
0
10000000
20000000
30000000
40000000
50000000
60000000

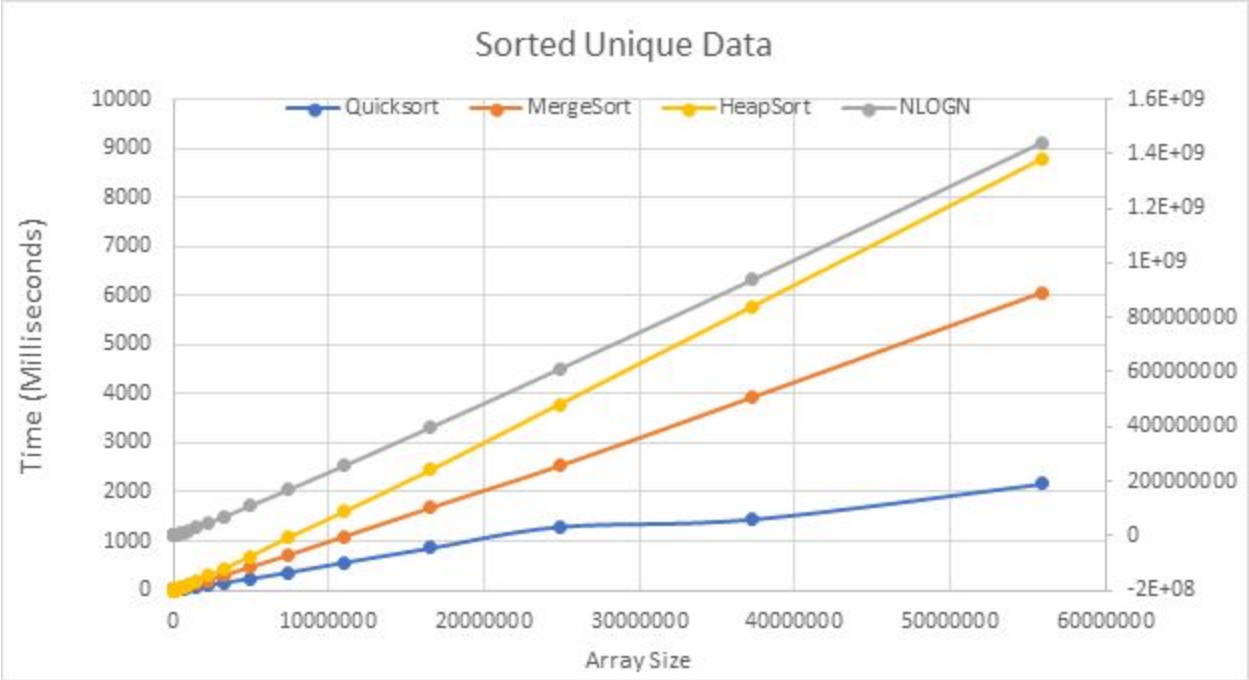

Almost Reverse Sorted Unique Data
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
12000
10000
8000
6000
4000
2000
0
1E+09
800000000
600000000
400000000
200000000
0
-2E+08
0
10000000
20000000
30000000
40000000

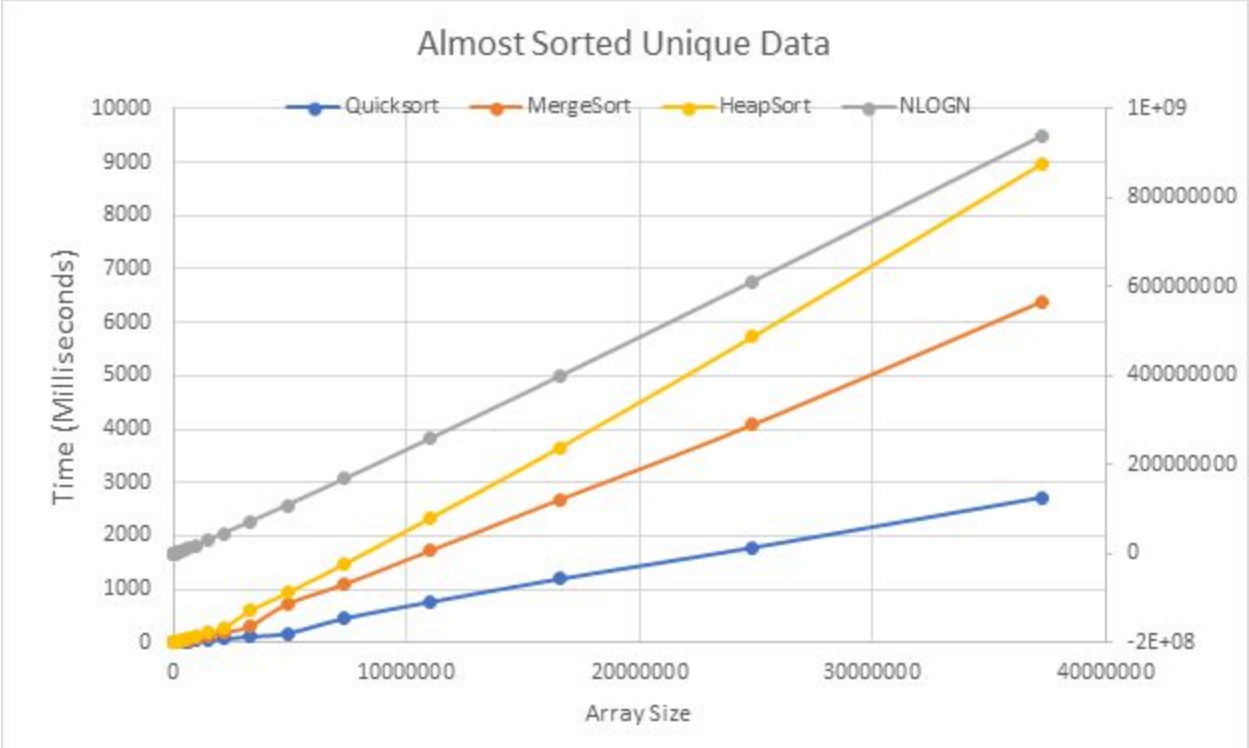

Random Unique Data
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
-1E+08
0
5000000
10000000
15000000
20000000
25000000
30000000

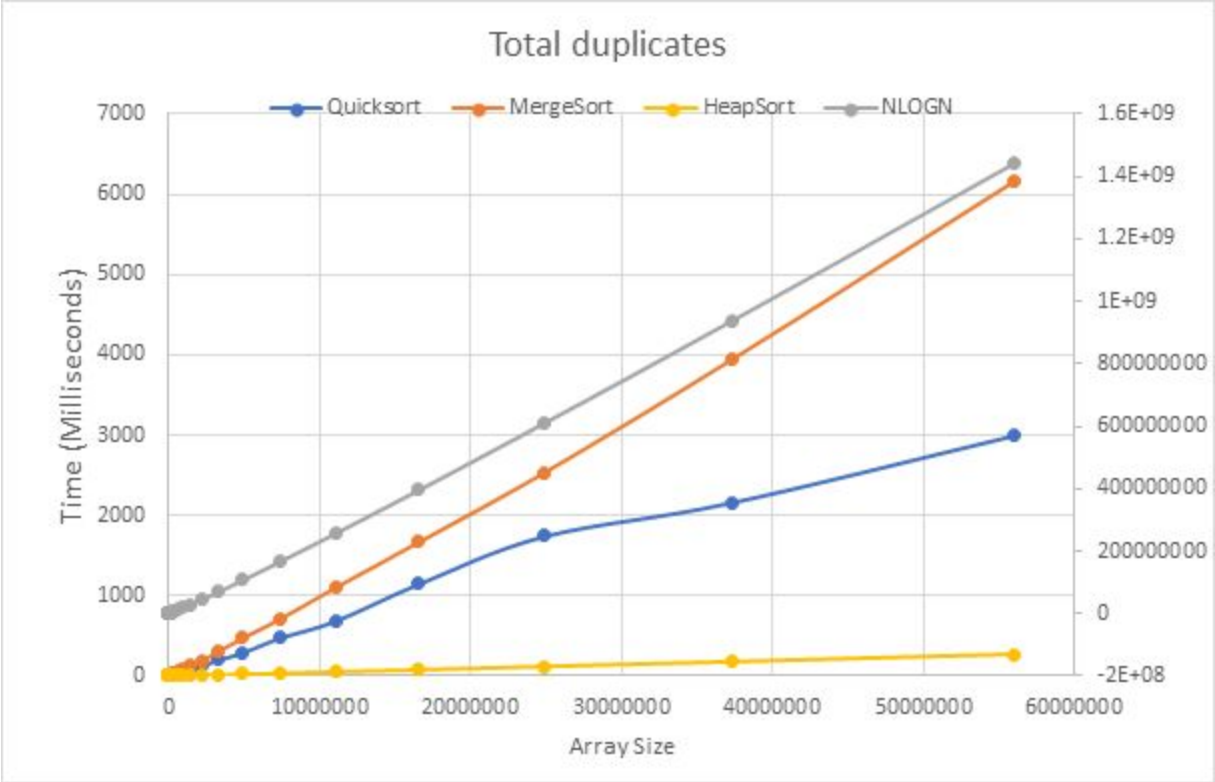

Random Duplicate Data
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
-1E+08
0
5000000
10000000
15000000
20000000
25000000
30000000

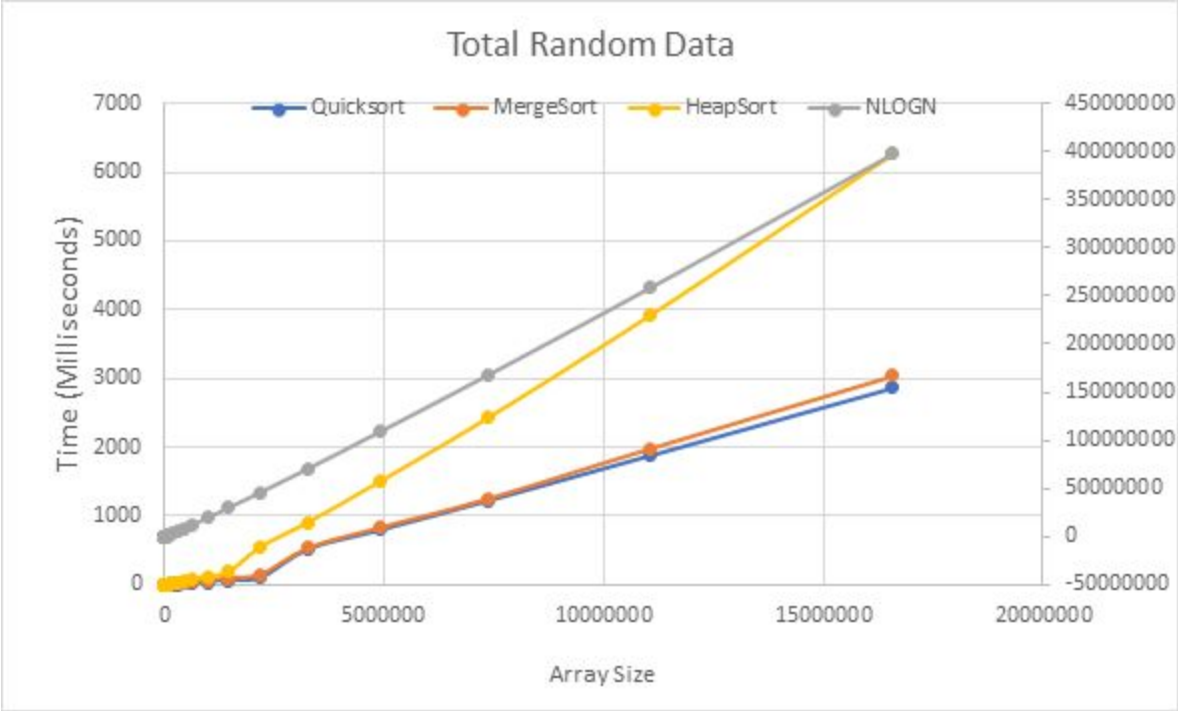

Sorted Duplicate Data
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1.6E+09
1.4E+09
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
-2E+08
0
10000000
20000000
30000000
40000000
50000000
60000000

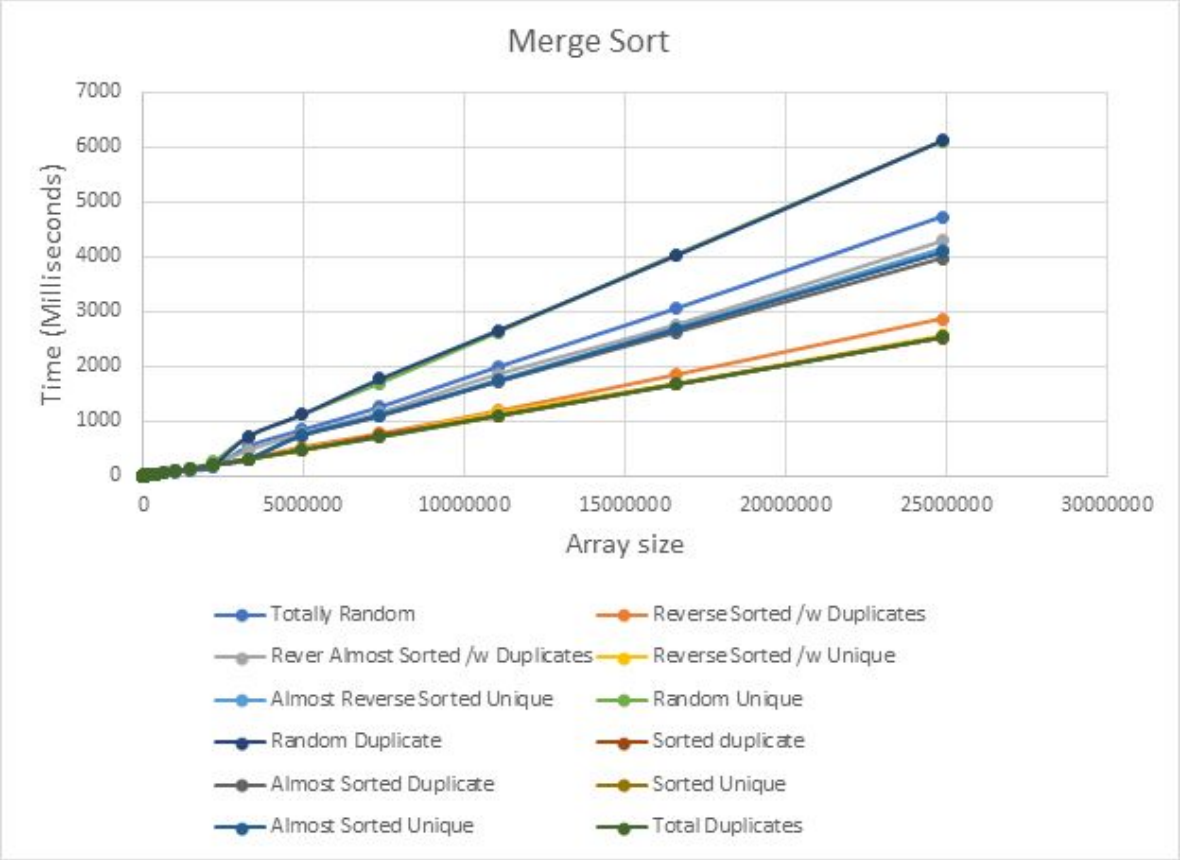

Almost Sorted Duplicate Data
MergeSort
HeapSort
Quicksort
NLOGN
Time (Milliseconds)
Array Size
7000
6000
5000
4000
3000
2000
1000
0
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
-1E+08
0
5000000
10000000
15000000
20000000
25000000
30000000

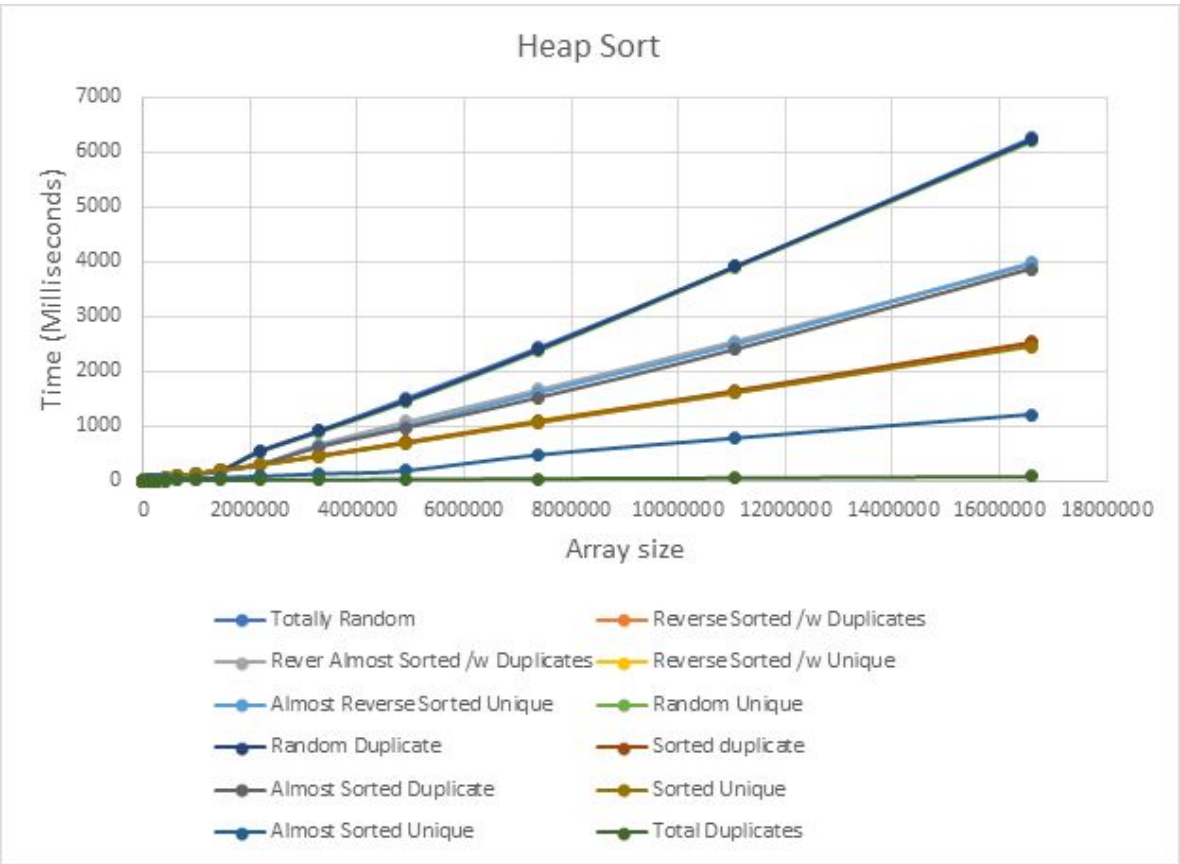

Sorted Unique Data
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1.6E+09
1.4E+09
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
-2E+08
0
10000000
20000000
30000000
40000000
50000000
60000000

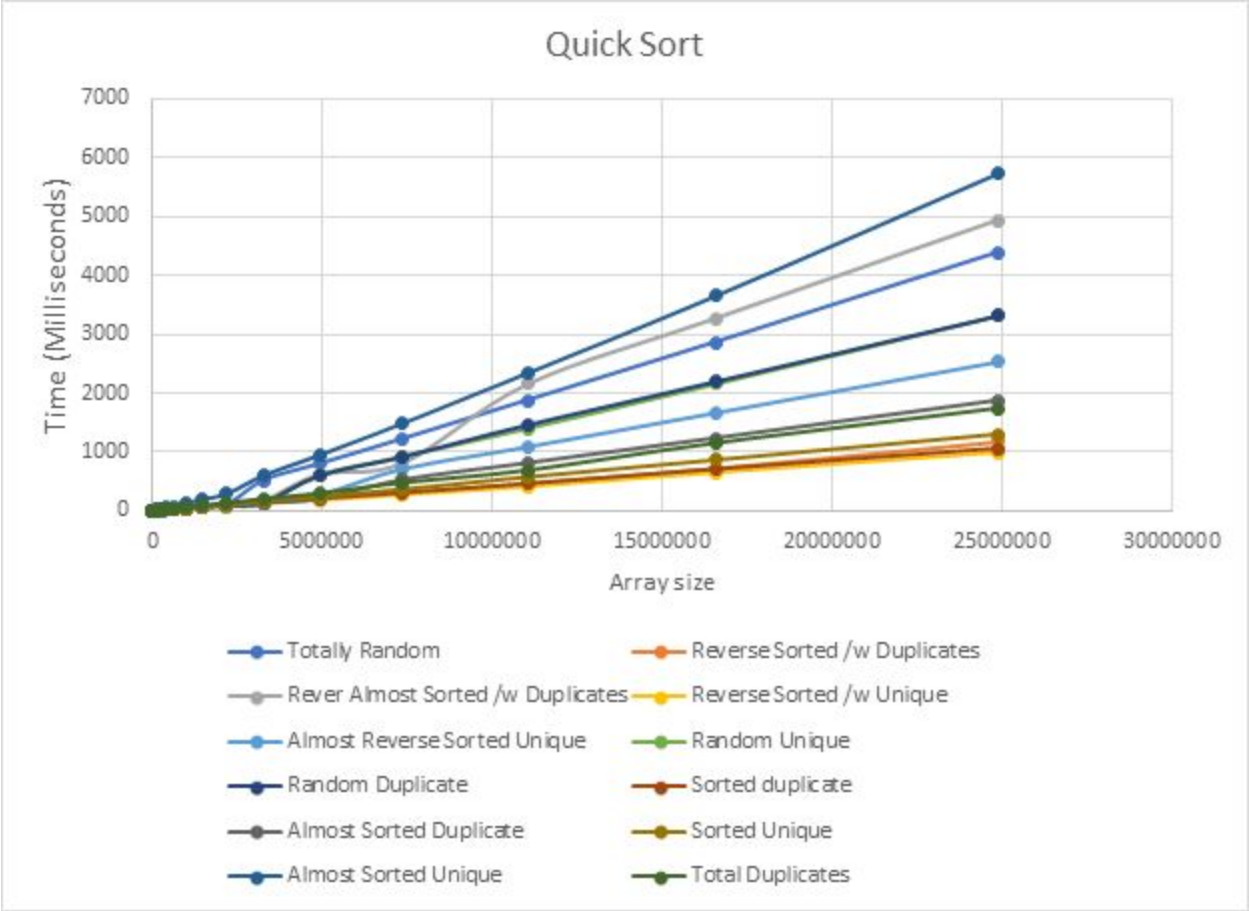

Almost Sorted Unique Data
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1E+09
800000000
600000000
400000000
200000000
0
-2E+08
0
10000000
20000000
30000000
40000000


Total duplicates
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
7000
6000
5000
4000
3000
2000
1000
0
1.6E+09
1.4E+09
1.2E+09
1E+09
800000000
600000000
400000000
200000000
0
-2E+08
0
10000000
20000000
30000000
40000000
50000000
60000000


Total Random Data
Quicksort
MergeSort
HeapSort
NLOGN
Time (Milliseconds)
Array Size
7000
6000
5000
4000
3000
2000
1000
0
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
-50000000
0
5000000
10000000
15000000
20000000


Merge Sort
Time (Milliseconds)
7000
6000
5000
4000
3000
2000
1000
0
0
5000000
10000000
15000000
20000000
25000000
30000000
Array size
Totally Random
Reverse Sorted /w Duplicates
Rever Almost Sorted /w Duplicates
Reverse Sorted /w Unique
Almost Reverse Sorted Unique
Random Unique
Random Duplicate
Sorted duplicate
Almost Sorted Duplicate
Sorted Unique
Almost Sorted Unique
Total Duplicates


Heap Sort
Time (Milliseconds)
7000
6000
5000
4000
3000
2000
1000
0
0
2000000
4000000
6000000
8000000
10000000
12000000
14000000
16000000
18000000
Array size
Totally Random
Reverse Sorted /w Duplicates
Rever Almost Sorted /w Duplicates
Reverse Sorted /w Unique
Almost Reverse Sorted Unique
Random Unique
Random Duplicate
Sorted duplicate
Almost Sorted Duplicate
Sorted Unique
Almost Sorted Unique
Total Duplicates


Quick Sort
Time (Milliseconds)
7000
6000
5000
4000
3000
2000
1000
0
0
5000000
10000000
15000000
20000000
25000000
30000000
Array size
Totally Random
Reverse Sorted /w Duplicates
Rever Almost Sorted /w Duplicates
Reverse Sorted /w Unique
Almost Reverse Sorted Unique
Random Unique
Random Duplicate
Sorted duplicate
Almost Sorted Duplicate
Sorted Unique
Almost Sorted Unique
Total Duplicates