Kordel France
Lab 2
Class 605.621 Section 83
3 August 2021

Lab 2 Analysis Document

FranceLab2 compares different algorithms in their ability to effectively sort data. This project is effectively an analysis of three different *quicksort* algorithms, but the performance of four additional algorithms was used to provide a more comprehensive comparison of how different upgrades on the original quicksort algorithm perform over different datasets. *FranceLab2* is a software module written in Python 3.7 that facilitates such algorithms[1].

Quicksort Median-of-Three Algorithm

A *quicksort median-of-three* algorithm is an augmented version of the original quicksort algorithm that selects the partitioning element as the middle of three specified values. Otherwise, for the purposes of the code in *FranceLab2,* the algorithm contains no other changes from original quicksort. Throughout this document, I will refer to *Quicksort Median-of-Three* as *QuicksortMOT.* The specification for this algorithm is below:

```
    1) /* This function is the driver code to implement the recursive quicksort median-of-three call
       (quicksortMOT).
       /*

function quicksortDriver( dataArray ) returns int [ ] {
        quicksortMOT ( dataArray, dataArray [ 0 ], dataArray [ dataArray.count – 1 ] )
        return dataArray
} endfunction


    2) /* This function is the main quicksort method. It is a recursive function that successively
       partitions the array according to the median-of-three principle.
       /*
function quicksortMOT( dataArray, startIndex, endIndex ) returns int [ ] {
        if (startIndex < endIndex ):
                partitionIndex = partitionArrayMOT ( dataArray, startIndex, endIndex )
                quicksortMOT( dataArray, startIndex, partitionIndex – 1, endIndex )
                quicksortMOT( dataArray, startIndex, partitionIndex + 1, endIndex )
        } endif
        return dataArray
} endfunction
```

[1] A table is included in Appendix A that shows the operational counts along with asymptotic regression equations for all seven algorithms side by side. It is essentially an extension of Table 1 for the other four sorting algorithms.

3) */* This function selects a partition index for the array according to the median-of-three principle: the middle of three values is selected as the partition index. Once the partition index is selected, a call is made to the regular quicksort partition algorithm.*
   */**

```
function partitionArrayMOT ( dataArray, startIndex, endIndex ) returns int [ ] {
    i_index = startIndex
    j_index = endIndex
    k_index = ( i_index – j_index / 2)
    pivotIndex = k_index
    if ( k_index < i_index and i_index < j_index ) {
        pivotIndex = i_index
    elif ( j_index < i_index and i_index < k_index ) {
        pivotIndex = i_index
    elif ( i_index < j_index and j_index < k_index ) {
        pivotIndex = j_index
    elif ( k_index < j_index and j_index < i_index ) {
        pivotIndex = j_index
    elif ( i_index < k_index and k_index < j_index ) {
        pivotIndex = k_index
    } endif
    exchange ( dataArray[startIndex], dataArray[pivotIndex] )
    return partitionArray ( dataArray, startIndex, endIndex )
} endfunction
```

4) */* This function partitions an array according to a designated starting and ending index. In the Lab 2 code, it is a common algorithm between all three quicksort algorithms.*
   */**

```
function partitionArray ( dataArray, startIndex, endIndex ) returns int {
    pivotIndex = startIndex
    pivotPoint = startIndex + 1
    for ( int i = startIndex + 1; i < endIndex + 1; i++ ) {
        if ( dataArray[ i ] <= dataArray [ pivotIndex ] ) {
            exchange ( dataArray [ pivotPoint ], dataArray [ i ] )
            pivotPoint += 1
        } endif
    } endloop
    exchange ( dataArray [ pivotIndex ], dataArray [ pivotPoint – 1 ] )
    pivotIndex = pivotPoint – 1
    return pivotIndex
} endfunction
```

Kordel France
Lab 2
Class 605.621 Section 83
3 August 2021

Asymptotic Behavior for *QuicksortMOT*

Quicksort has two operations that make it particularly easy to analyze its runtime for – comparisons and exchanges. As a consequence, *QuicksortMOT* is no different in this regard. Below we will discuss the asymptotic behavior of *QuicksortMOT* with a particular focus on the number of comparisons and exchanges made by the algorithm as a key contributor to its asymptotic cost.

1. To begin, the driver function *quicksortDriver* makes a call to the primary sorting algorithm. Since this is a driver function, we do not consider any costs of this function as part of the final costs for *QuicksortMOT.*

   Cost: ***O (0)***

2. The *quicksortMOT* function accepts an argument of the data to be sorted. We verify the starting index is smaller than the ending index and then we partition the data array.

   Cost: ***O (c)***

3. The function *partitionArrayMOT* is called next. This function gives *QuicksortMOT* its name by selecting the middle of three values: the starting index, the ending index, and a combination of the two values. Once this partitioning point is established, the data array is passed to the *partitionArray* function where the actual partitioning of the data array occurs.

   Cost: ***O (c)***

4. Next, we consider the *partitionArray* function. This function actually performs the sorting from one index to another based on the pivot index provided in Step 3. **This function contains the operations that are the highest contributors to the runtime.** We start off by establishing pivot indices which take *O(c)* time. The *for-loop* inside the function contains a conditional *if-statement* that brackets an exchange and comparison operation. This conditional statement prevents us from applying a direct *O (n)* runtime on the loop. While the loop will be performed *n* times, the operations inside are conditional on the pivot index and loop index. At a worst-case scenario, we expect the entire data array to be sorted in reverse order, in which case the statements within the loop will all execute *n* times resulting in *n* comparisons and *n* exchanges. At a best-case scenario, the data array is already sorted, so we neither comparisons nor exchanges are performed. In an average scenario, we expect *half* the values to be sorted, resulting in *n / 2* exchanges and comparisons each. After exiting the loop, we have a final exchange to move the pivot forward and return the next partition index.

   Best Case Cost: ***O (c)***
   Average Case Cost: ***O (n / 2)***
   Worst Case Cost: ***O (n)***

5. Now that the partition index is established, we are back inside the *quicksortMOT* function. We use the received index to divide the array into two parts – one on either side of the partitioning index. This warrants two recursion calls to *quicksortMOT* (one for either side of the partitioning index) and steps 2 – 4 are repeated. This process continues until the conditional statement of *quicksortMOT* is violated, after which the newly sorted array is returned. The recursion stack is *n* deep because *quicksortMOT* is called *n* times. Since *quicksortMOT* envelopes the operations from *partitionArrayMOT* and *partitionArray*. We can multiply the runtimes received in steps 2 – 4 by 2. This is straightforward for best and worst case datasets, but for an average runtime, *partitionArray* is encompassing ½ of the data. So *n* recursion calls to half of the dataset will result in $((((n/2)/2)/\dots/2)$ which converges to $\lg n$ for an average case. Based on this, we receive the following.

Best Case Cost: $T(n) = O(n) * [O(0) + O(c) + O(c) + O(c)]$
$= O(n) * 3 * O(c)$
$= O(n * 3c)$
$\Rightarrow O(n)$

Average Case Cost: $T(n) = O(n) * [O(0) + O(c) + O(c) + O(\lg n)]$
$= O(n) * (O(2c) + O(\lg n)]$
$= O(n * [2c + \lg n])$
$\Rightarrow O(n \lg n)$

Worst Case Cost: $T(n) = O(n) * [O(0) + O(c) + O(c) + O(n)]$
$= O(n) * [2 * O(c) + O(n)]$
$= O(n * [2c + n])$
$\Rightarrow O(n^2)$

One might notice that the runtimes above for *QuicksortMOT* are exactly that of regular vanilla quicksort. Although this is true, the strategy behind the *median-of-three* principle is that it allows the algorithm to *approach* average case runtimes on worst case datasets. With regular quicksort, the pivot point is not selected with a heuristic. With the *median-of-three* strategy, the pivot index is chosen to optimize the number of partitions that actually occur. This strategy gives a better estimate of an optimal pivot index when no information about the dataset is known (sorted, unsorted, reverse sorted, etc.). This may be thought of as a probabilistic advantage that increases the odds that a better index is chosen over selecting any single index. Due to this, it is probably adequate to say that $O(n^2)$ is only an upper bound (and a much more loose upper bound) on the runtime of *QuicksortMOT* and not an actual possible runtime. Simply put, selecting the median of three indices allows the regular quicksort algorithm to avoid a worst-case runtime on a worst-case dataset.

Kordel France
Lab 2
Class 605.621 Section 83
3 August 2021

Cost Analysis for All Quicksort Algorithms

Space Complexity

All three algorithms were developed to be identical other than the partition strategy; all three algorithms were also developed using recursion instead of iteration. Due to this, the *difference* in space complexity between the three algorithms is a function of the depth of the recursion stack. Less recursive calls will result to a smaller stack and, consequently, less subarrays being stored in memory. The recursion stack is lower for *quicksortMOT* on worst case datasets in comparison to regular *quicksort* because the former is optimized specifically for worst-case datasets, but *quicksortMOT* has a slightly higher recursion stack on best-case datasets. The space complexity, in general, is a function of the recursion stack and is dependent on both algorithm and dataset.

Time Complexity

Please refer to the table below for acquired runtime costs for both algorithms in the *FranceLab2* module. The table shows the number of exchanges and comparisons performed for each algorithm on each of the three datasets for different sizes of *n*. In order to more accurately assess just how fast the number of operations scaled with respect to changes in *n*, I fit exponential regression curves of the form *y = ax^b* to each algorithm over each dataset. The last two columns in the table give these equations to help validate the asymptotic trajectory of each algorithm. Please reference `Table 1` below for the following sections.

#	data type	n	algorithm	comparisons	exchanges	comparison trajectory equation	exchange trajectory equation
1	sorted	50	quicksort	0	49	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
2	sorted	500	quicksort	0	499	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
3	sorted	1000	quicksort	0	999	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
4	sorted	1500	quicksort	0	1499	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
5	sorted	2000	quicksort	0	1999	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
6	sorted	50	quicksort_rand	82	114	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
7	sorted	500	quicksort_rand	2654	2975	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
8	sorted	1000	quicksort_rand	4756	5398	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
9	sorted	1500	quicksort_rand	7663	8611	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
10	sorted	2000	quicksort_rand	10978	12241	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
11	sorted	50	quicksort_mot	86	132	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
12	sorted	500	quicksort_mot	977	1470	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
13	sorted	1000	quicksort_mot	1975	2967	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
14	sorted	1500	quicksort_mot	2973	4463	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
15	sorted	2000	quicksort_mot	3973	5964	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
16	reverse	50	quicksort	625	674	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
17	reverse	500	quicksort	62500	62999	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
18	reverse	1000	quicksort	250000	250999	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
19	reverse	1500	quicksort	562500	563999	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
20	reverse	2000	quicksort	1000000	1001999	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
21	reverse	50	quicksort_rand	129	164	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
22	reverse	500	quicksort_rand	1916	2257	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
23	reverse	1000	quicksort_rand	4917	5606	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
24	reverse	1500	quicksort_rand	8775	9788	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
25	reverse	2000	quicksort_rand	11780	13166	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
26	reverse	50	quicksort_mot	245	289	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
27	reverse	500	quicksort_mot	20856	21340	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
28	reverse	1000	quicksort_mot	83348	84330	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
29	reverse	1500	quicksort_mot	187287	188768	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
30	reverse	2000	quicksort_mot	333166	335147	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
31	random	50	quicksort	156	188	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
32	random	500	quicksort	2483	2813	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
33	random	1000	quicksort	5988	6659	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
34	random	1500	quicksort	9264	10259	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
35	random	2000	quicksort	12871	14199	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
36	random	50	quicksort_rand	132	162	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
37	random	500	quicksort_rand	2507	2838	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
38	random	1000	quicksort_rand	5445	6113	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
39	random	1500	quicksort_rand	9843	10847	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
40	random	2000	quicksort_rand	13870	15194	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
41	random	50	quicksort_mot	171	204	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
42	random	500	quicksort_mot	2922	3266	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
43	random	1000	quicksort_mot	5951	6629	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
44	random	1500	quicksort_mot	9758	10748	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
45	random	2000	quicksort_mot	14842	16182	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %

Table 1 - Actual costs with expected asymptotic costs fitted through exponential regression for the three quicksort algorithms.

Sorted Data

One will notice that the regression equations roughly show $y = x^1 = x \rightarrow n$ for the regular *quicksort* and *quickortMOT* algorithms over sorted data (lines 1 – 15). A sorted dataset provides the best case runtime and these values validate our proof above showing that the asymptotic cost of *quicksort* and *quicksortMOT* trends towards *O (n).* As a check, we can add the number of comparisons and exchanges for lines 1 and 11 in the table which shows that there were 86 + 132 = 218 operations performed for *quicksortMOT,* and 0 + 49 = 49 operations performed for regular *quicksort* respectively over n = 50. Since both of these resultant values are constant multiples of n, the expected runtime of *O (n)* holds. *Randomized quicksort* shows a slightly higher cost than its counterparts which is expected over the sorted dataset since the randomized partitioning essentially "unsorts" the already sorted data to some degree resulting in more operations. As such, the exponents in its regression equations are slightly higher than that of the other two algorithms.

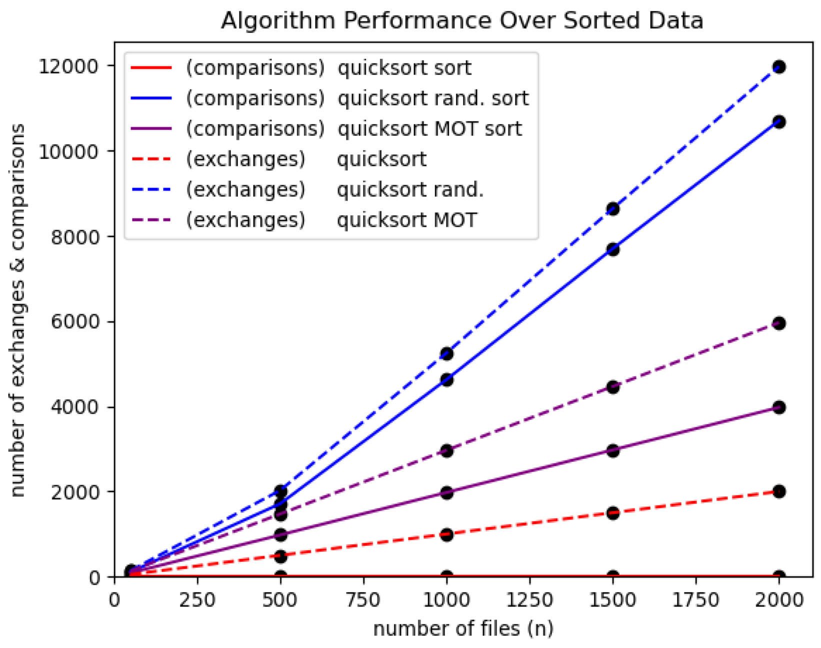

Figure 1 - A display of the asymptotic growth of the three quicksort algorithms over a sorted dataset for successively larger values of n.

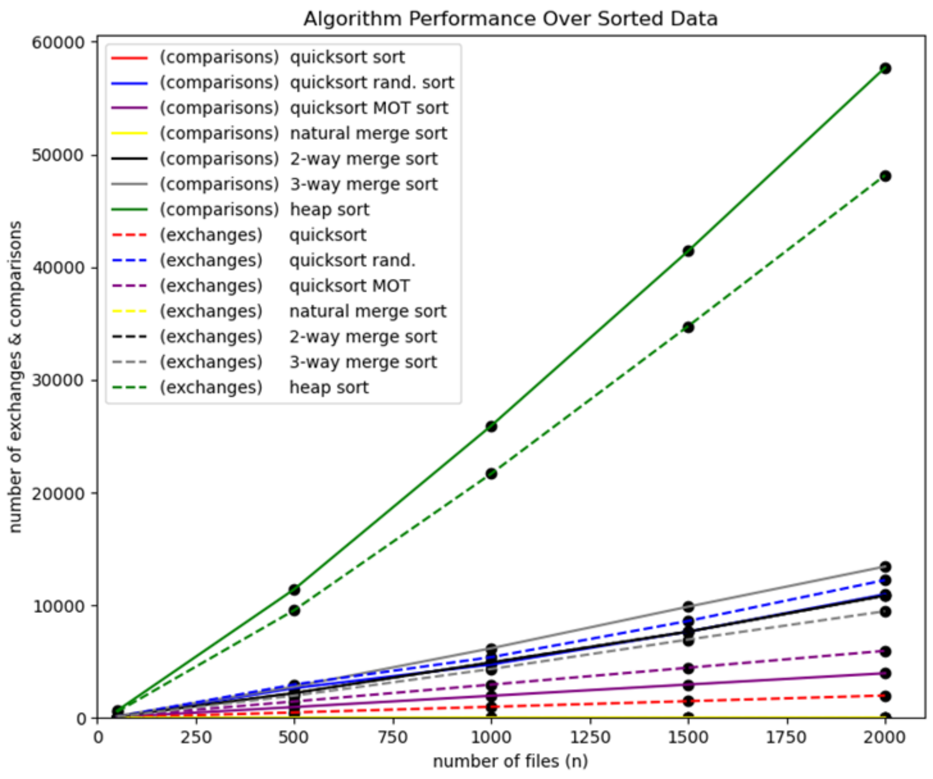

Figure 2 - A display of the asymptotic growth of the three quicksort algorithms against four other sorting algorithms over a sorted dataset for successively larger values of n.

Reverse Sorted Data

The worst case reverse sorted dataset is where we really see the benefit of the *median-of-three* strategy. For regular quicksort, we observe that the regression equations for regular *quicksort* show exponents of roughly $y = x^{1.81} \rightarrow x^2 \rightarrow n^2$ for both the number of comparisons and number of exchanges. Once again, this validates our proof above showing that regular *quicksort* has an asymptotic cost of $O(n^2)$. Interestingly enough, *quicksortMOT* shows slightly slower growth of $y = x^{1.65}$ which is expected since the *median-of-three* strategy is specifically intended to benefit worst datasets. As a check, we can add the number of comparisons and exchanges for lines 16 and 26 in the table which shows that there were 625 + 674 = 1250 operations performed for *quicksort*, and 245 + 289 = 534 operations performed for regular *quicksortMOT* respectively over n = 50. Since both of these resultant values are constant multiples of n^2 and n^2 provides an upper bound on these values, the expected runtime of $O(n^2)$ holds. *Randomized quicksort* shows the best runtime of any of the algorithms over this worst case dataset, which is slightly misleading. While less operations and exchanges were technically performed under the randomized partitioning strategy, we don't account for the cost of the random number generator. The reason for this is because it is difficult to account for without knowing the algorithm behind the random number generator, but it provides operational debt for the algorithm so it needs to be considered. Due to this the actual asymptotic costs of *randomized quicksort* are slightly higher than the table suggests.

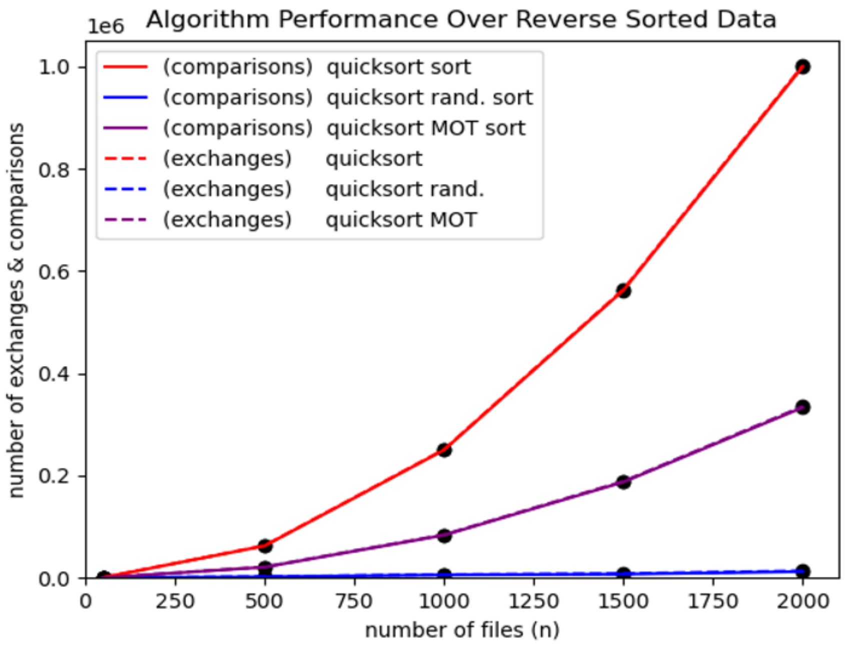

Figure 3 - A display of the asymptotic growth of the three quicksort algorithms over a reverse sorted dataset for successively larger values of n.

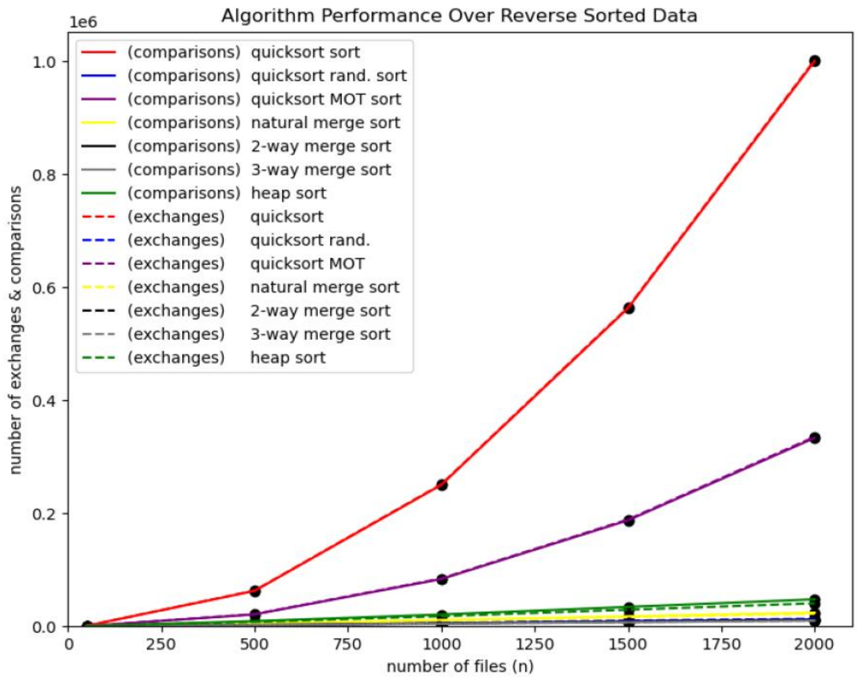

Figure 4 - A display of the asymptotic growth of the three quicksort algorithms against four other sorting algorithms over a reverse sorted dataset for successively larger values of n.

Randomized Data

For the finale, we address the average case over a randomized dataset. As a method of control, all three algorithms sorted the same dataset for equal sizes of *n*. One will notice that there is a general decrease in variance among the numbers of operations and exchanges for the algorithms for identical sizes of *n*. For all three quicksort algorithms, we see regression functions all within the neighborhood of $y = x^{1.25} \rightarrow n^{1.25}$ and $y = x^{1.27} \rightarrow n^{1.27}$ for the numbers of comparisons and exchanges respectively. From our proof above, we expect the average case runtime of these algorithms over randomized datasets to be *O (n lg n).* We can check that this condition is satisfied by seeing that for *n* = 50, we expect the runtime to be *n lg n = 50 * lg(50) = 50 * 5.64 = 282.* If we add the number of comparisons and operations for all three algorithms at *n = 50*, we receive *344*, *294*, and *375* for *quicksort*, *randomized quicksort*, *and quicksortMOT* respectively. All three numbers are relatively close to *n lg n* so we accept that the expected value of *O (n lg n)* holds, especially because *n lg n* is not an upper or lower bound on the expected runtime. We can show further accuracy by averaging the cost of the algorithm over all instances of *n* and over multiple datasets, which is shown in *Appendix A.* It's worth noting again that the realized costs of *randomized quicksort* are slightly higher due to the cost to randomize. Once again, this validates our proof above.

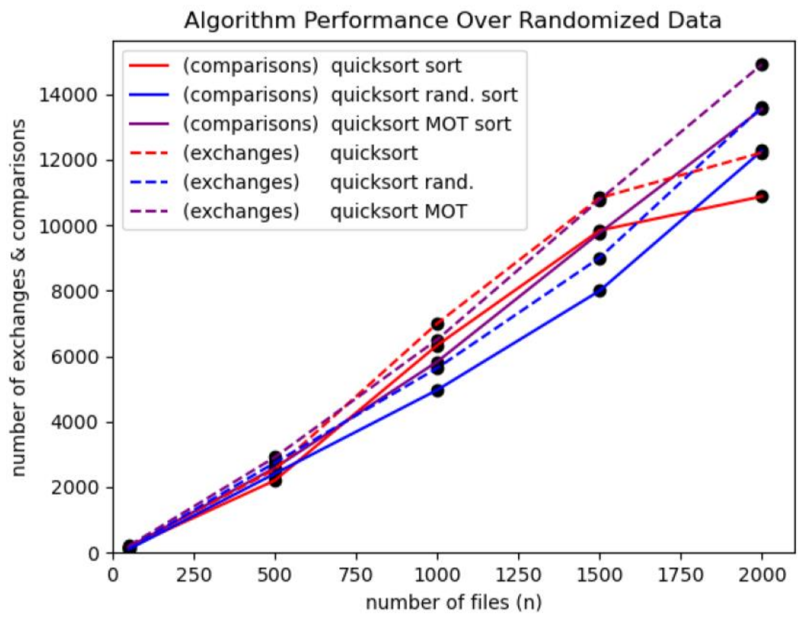

Figure 5 - A display of the asymptotic growth of the three quicksort algorithms over a randomized dataset for successively larger values of n.

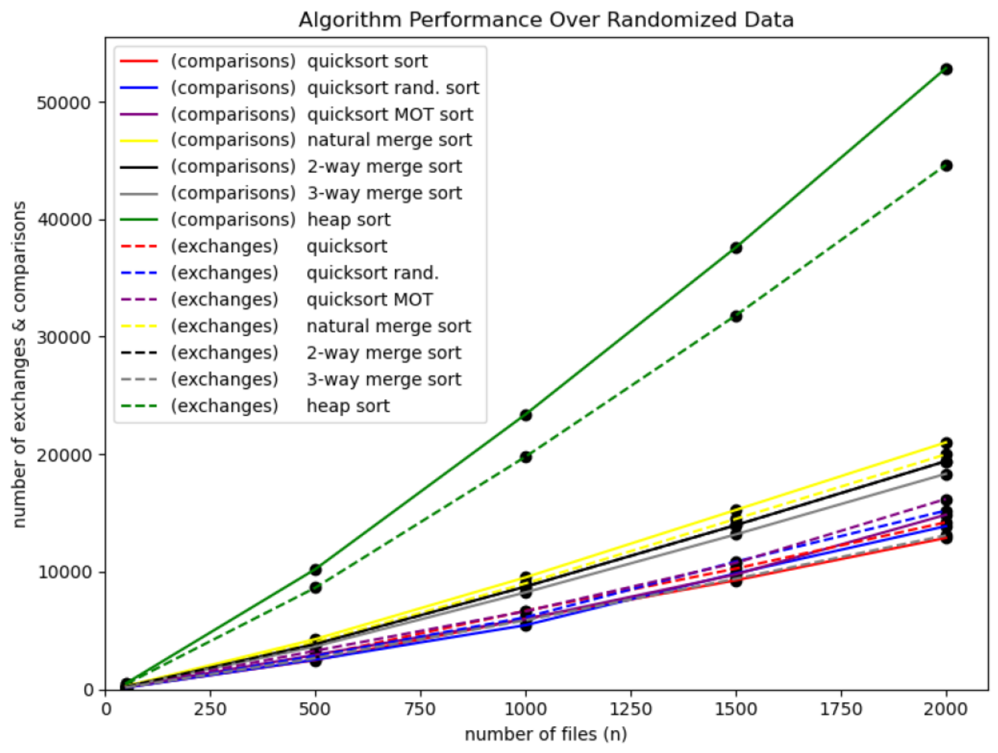

Figure 6 - A display of the asymptotic growth of the three quicksort algorithms against four other sorting algorithms over a randomly sorted dataset for successively larger values of n.

A table is included in *Appendix A* that shows the operational counts along with asymptotic regression equations for all seven algorithms side by side. It is essentially an extension of Table 1 for the other four sorting algorithms.

Optimizations for Later Versions

As specified previously, I would choose to implement at least two of my algorithms utilizing tail recursion. Additionally, my code contains a mix of camel case (camelCase) and "python_style" (python_style) syntax which I would like to clean up. Working on the project across multiple periods while working across multiple languages with work is the driver behind the inconsistency. I would also optimize the time complexity of the driver code in *__main__.py*. Finally, I would like to alter *FranceLab2* to assess *k-way* partitioning for different values of *k* for a few quicksort algorithms similar to how *k-way* merging is assessed for mergesort.

Enhancements

The program provides 8 major enhancements (along with many other minor enhancements) on top of the original requirements. These enhancements are also elaborated upon in the *.README* document associated with the module

1) Time Delays - processing is paused briefly throughout the program to allow the user time to read and interpret the output. This creates for a much better user experience.

2) Execution time is tracked and monitored for each sorting algorithm.

3) Each sorting run is graphed in a `*.csv*` file. Files are named {algorithm_name}-{date_type}-{n} count.csv such as *'quicksortMOT_random_1000count.csv'*. Each file is located in the *output_files* directory and contains the following properties:
   - the data type.
   - file count.
   - sorting algorithm name.
   - number of comparisons performed.
   - number of exchanges performed.
   - an equation that plots the trajectory of the number of comparisons made by the algorithm over this data type as `n` scales along with the correlation coefficient between the equation and the observed data.
   - an equation that plots the trajectory of the number of exchanges made by the algorithm over this data type as `n` scales along with the correlation coefficient between the equation and the observed data.
   - the initial dataset for each run before it entered the sorting algorithm.
   - the final dataset for each run after it entered the sorting algorithm and was fully sorted.
   - the execution time (in seconds) for the algorithm to completely sort the data.

4) A `Summary Table` is provided at the very end of the program that shows the performance of each algorithm over different data distributions. A `*FINAL_ANALYSIS.csv*` file is a direct copy of the `*Summary Table*`, but in a `*.csv*` file that shows performance over all 72 sorting runs.

5) Equations for trajectory curves were calculated to extrapolate the number of comparisons \ exchanges that would be theoretically needed for very large *n* as the algorithm scales. This was accomplished by calculating coefficients of power regression to define the trajectory path based on the data gathered for similar sorts. If one opens *Metric.py* where the regression algorithms are located, they will notice the regression curve is computed from scratch with no "packages" - the regression equation is derived from low-level statistics functions. The `*numpy*` package is only used to cast an array type to one of a type easier to manipulate with these statistics functions.

6) Correlation values are calculated (again from scratch and without any use of packages) to show how well the above regression curve fits the empirically gathered data in our analysis.

7) A status is communicated to the user as a % complete in the `*__main__.py*` file while the data is being processed and sorted. This allows for a more appealing user interface and lets the user have an idea of where the program is at in its execution steps.

8) Plots of all of the data runs for each algorithm are shown and allowed for easy comparison against other algorithms. This makes it simple for the user to spot analytical trends and spot which algorithms out-perform others on certain datasets.

Lessons Learned

This lab showed me how simple changes to the same algorithm can allow the algorithm to "specialize" in sorting certain datasets faster. It was interesting to see how the *median-of-three* strategy allowed *quicksort* to provide better runtimes over worst case datasets but slightly worse runtimes over best case datasets in comparison to its counterparts. This goes to show that, if we can have any prior knowledge about the dataset we are sorting, this can help us determine the most efficient algorithm to use. If the best case runtime is highly improbable to encounter in an application then it doesn't make much sense to use vanilla quicksort as the sorting algorithm.

Through this lab, I found that, at least for sorting algorithms, counting the number of comparisons and the number of exchanges performed by an algorithm gives a very good indication of the runtime complexity of the algorithm. By counting just these two operations, I was able to provide very accurate estimates of asymptotic costs. This hypothesis seems to be consistent among all sorting algorithms because I applied a similar approach with the other four *mergesort* and *heapsort* algorithms. The numbers of exchanges and operations alone seem to be great heuristics for assessing costs in sorting algorithms.

All in all, *FranceLab2* provided me with another angle of appreciation for recursion. Using recursion provided a form of experimental control over the three quicksort algorithms by allowing me to have a common algorithmic structure as well as a common baseline to determine asymptotic costs for both time and space. It would be interesting to see how the costs of the quicksort algorithms changed with iterative implementations in later a later version of the program.

Conclusion

FranceLab2 addresses the requirements as specified in the *Lab 2* handout. Specifically, the program generates 3 different datasets over 5 different counts and inputs each of those into 7 different algorithms for a grand total of 105 trace runs. The data is sorted, graphed, and output to the user with metrics on algorithmic efficiency for each run. Finally, the user is presented with a summary for all data runs performed and the program is terminated.

References

The following items were used as references for the construction of this project.

1) Cormen, T. H., & Leiserson, C. E. (2009). *Introduction to Algorithms, 3rd edition.*

2) Miller, B. N., & Ranum, D. L. (2014). *Problem solving with algorithms and data structures using Python (2nd ed.).* Decorah, IA: Brad Miller, David Ranum

3) Merge Sort. GeeksforGeeks. (2021, June 20). https://www.geeksforgeeks.org/merge-sort/.

4) Rayapati, P. (2019, August 20). *3-Way merge sort.* Retrieved April 25, 2021, from https://www.geeksforgeeks.org/3-way-merge-sort/.

5) Woltmann, S. (2021, January 13). *Merge sort – ALGORITHM, source code, time complexity.* Retrieved April 24, 2021, from https://www.happycoders.eu/algorithms/merge-sort/#Natural_Merge_Sort.

6) *K-way merge algorithm.* (2021, April 09). Retrieved April 24-27, 2021, from https://en.wikipedia.org/wiki/K-way_merge_algorithm

7) *Artificial Intelligence: A Modern Approach. Third Edition*. Russel, Stuart J.; Norvig, Peter. 2015, Pearson India Education Services Pvt. Ltd. p 961-962.

8) *Deep Learning*. Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. 2016, Massachusetts Institute of Technology. p 147 -149, 525 – 527.

9) Garey, M. R., & Johnson, D. S. (2003). *Computers and Intractability: A Guide to the Theory of NP - Completeness.* W.H. Freeman and Co.

10) Kleinberg, J., & Tardos, É. (2014). *Algorithm Design.* Pearson India Education Services Pvt Ltd.

Appendix A

#	data_type	n	algorithm	comparisons	exchanges	comparison trajectory equation	exchange trajectory equation
1	sorted	50	quicksort	0	49	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
2	sorted	500	quicksort	0	499	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
3	sorted	1000	quicksort	0	999	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
4	sorted	1500	quicksort	0	1499	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
5	sorted	2000	quicksort	0	1999	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
6	sorted	50	quicksort_rand	82	114	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
7	sorted	500	quicksort_rand	2654	2975	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
8	sorted	1000	quicksort_rand	4756	5398	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
9	sorted	1500	quicksort_rand	7663	8611	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
10	sorted	2000	quicksort_rand	10978	12241	y = 1.0006 (x) ^ 1.2242 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2412 w/ correlation 100.0 %
11	sorted	50	quicksort_mot	86	132	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
12	sorted	500	quicksort_mot	977	1470	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
13	sorted	1000	quicksort_mot	1975	2967	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
14	sorted	1500	quicksort_mot	2973	4463	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
15	sorted	2000	quicksort_mot	3973	5964	y = 1.001 (x) ^ 1.0957 w/ correlation 100.0 %	y = 1.0017 (x) ^ 1.1528 w/ correlation 100.0 %
16	sorted	50	natural_merge	1	0	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
17	sorted	500	natural_merge	1	0	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
18	sorted	1000	natural_merge	1	0	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
19	sorted	1500	natural_merge	1	0	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
20	sorted	2000	natural_merge	1	0	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %	y = 1.0 (x) ^ 0.9999 w/ correlation 100.0 %
21	sorted	50	2-way_merge	133	133	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %
22	sorted	500	2-way_merge	2216	2216	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %
23	sorted	1000	2-way_merge	4932	4932	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %
24	sorted	1500	2-way_merge	7664	7664	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %
25	sorted	2000	2-way_merge	10864	10864	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.2266 w/ correlation 100.0 %
26	sorted	50	3-way_merge	171	126	y = 1.0012 (x) ^ 1.2604 w/ correlation 100.0 %	y = 1.0006 (x) ^ 1.2112 w/ correlation 100.0 %
27	sorted	500	3-way_merge	2790	1986	y = 1.0012 (x) ^ 1.2604 w/ correlation 100.0 %	y = 1.0006 (x) ^ 1.2112 w/ correlation 100.0 %
28	sorted	1000	3-way_merge	6187	4352	y = 1.0012 (x) ^ 1.2604 w/ correlation 100.0 %	y = 1.0006 (x) ^ 1.2112 w/ correlation 100.0 %
29	sorted	1500	3-way_merge	9877	6963	y = 1.0012 (x) ^ 1.2604 w/ correlation 100.0 %	y = 1.0006 (x) ^ 1.2112 w/ correlation 100.0 %
30	sorted	2000	3-way_merge	13441	9476	y = 1.0012 (x) ^ 1.2604 w/ correlation 100.0 %	y = 1.0006 (x) ^ 1.2112 w/ correlation 100.0 %
31	sorted	50	heap_sort	644	543	y = 1.0035 (x) ^ 1.4615 w/ correlation 100.0 %	y = 1.0031 (x) ^ 1.4367 w/ correlation 100.0 %
32	sorted	500	heap_sort	11424	9567	y = 1.0035 (x) ^ 1.4615 w/ correlation 100.0 %	y = 1.0031 (x) ^ 1.4367 w/ correlation 100.0 %
33	sorted	1000	heap_sort	25926	21672	y = 1.0035 (x) ^ 1.4615 w/ correlation 100.0 %	y = 1.0031 (x) ^ 1.4367 w/ correlation 100.0 %
34	sorted	1500	heap_sort	41426	34762	y = 1.0035 (x) ^ 1.4615 w/ correlation 100.0 %	y = 1.0031 (x) ^ 1.4367 w/ correlation 100.0 %
35	sorted	2000	heap_sort	57700	48151	y = 1.0035 (x) ^ 1.4615 w/ correlation 100.0 %	y = 1.0031 (x) ^ 1.4367 w/ correlation 100.0 %
36	reverse	50	quicksort	625	674	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
37	reverse	500	quicksort	62500	62999	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
38	reverse	1000	quicksort	250000	250999	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
39	reverse	1500	quicksort	562500	563999	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
40	reverse	2000	quicksort	1000000	1001999	y = 0.9978 (x) ^ 1.8055 w/ correlation 100.0 %	y = 0.9978 (x) ^ 1.806 w/ correlation 100.0 %
41	reverse	50	quicksort_rand	129	164	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
42	reverse	500	quicksort_rand	1916	2257	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
43	reverse	1000	quicksort_rand	4917	5606	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
44	reverse	1500	quicksort_rand	8775	9788	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %
45	reverse	2000	quicksort_rand	11780	13166	y = 0.9978 (x) ^ 1.2367 w/ correlation 100.0 %	y = 0.9986 (x) ^ 1.2534 w/ correlation 100.0 %

46	reverse	50	quicksort_mot	245	289	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
47	reverse	500	quicksort_mot	20856	21340	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
48	reverse	1000	quicksort_mot	83348	84330	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
49	reverse	1500	quicksort_mot	187287	188768	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
50	reverse	2000	quicksort_mot	333166	335147	y = 0.996 (x) ^ 1.6513 w/ correlation 100.0 %	y = 0.9961 (x) ^ 1.6527 w/ correlation 100.0 %
51	reverse	50	natural_merge	344	294	y = 1.0022 (x) ^ 1.3408 w/ correlation 100.0 %	y = 1.0019 (x) ^ 1.3278 w/ correlation 100.0 %
52	reverse	500	natural_merge	4992	4492	y = 1.0022 (x) ^ 1.3408 w/ correlation 100.0 %	y = 1.0019 (x) ^ 1.3278 w/ correlation 100.0 %
53	reverse	1000	natural_merge	10984	9984	y = 1.0022 (x) ^ 1.3408 w/ correlation 100.0 %	y = 1.0019 (x) ^ 1.3278 w/ correlation 100.0 %
54	reverse	1500	natural_merge	17492	15992	y = 1.0022 (x) ^ 1.3408 w/ correlation 100.0 %	y = 1.0019 (x) ^ 1.3278 w/ correlation 100.0 %
55	reverse	2000	natural_merge	23968	21968	y = 1.0022 (x) ^ 1.3408 w/ correlation 100.0 %	y = 1.0019 (x) ^ 1.3278 w/ correlation 100.0 %
56	reverse	50	2-way_merge	153	153	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %
57	reverse	500	2-way_merge	2272	2272	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %
58	reverse	1000	2-way_merge	5044	5044	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %
59	reverse	1500	2-way_merge	8288	8288	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %
60	reverse	2000	2-way_merge	11088	11088	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %	y = 1.0001 (x) ^ 1.234 w/ correlation 100.0 %
61	reverse	50	3-way_merge	133	133	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %
62	reverse	500	3-way_merge	2021	2021	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %
63	reverse	1000	3-way_merge	4436	4436	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %
64	reverse	1500	3-way_merge	7060	7060	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %
65	reverse	2000	3-way_merge	9848	9848	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %	y = 1.0008 (x) ^ 1.2135 w/ correlation 100.0 %
66	reverse	50	heap_sort	430	373	y = 1.0023 (x) ^ 1.4322 w/ correlation 100.0 %	y = 1.0021 (x) ^ 1.4099 w/ correlation 100.0 %
67	reverse	500	heap_sort	8968	7661	y = 1.0023 (x) ^ 1.4322 w/ correlation 100.0 %	y = 1.0021 (x) ^ 1.4099 w/ correlation 100.0 %
68	reverse	1000	heap_sort	20680	17657	y = 1.0023 (x) ^ 1.4322 w/ correlation 100.0 %	y = 1.0021 (x) ^ 1.4099 w/ correlation 100.0 %
69	reverse	1500	heap_sort	34102	29074	y = 1.0023 (x) ^ 1.4322 w/ correlation 100.0 %	y = 1.0021 (x) ^ 1.4099 w/ correlation 100.0 %
70	reverse	2000	heap_sort	47568	40493	y = 1.0023 (x) ^ 1.4322 w/ correlation 100.0 %	y = 1.0021 (x) ^ 1.4099 w/ correlation 100.0 %
71	random⍰	50	quicksort	156	188	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
72	random⍰	500	quicksort	2483	2813	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
73	random⍰	1000	quicksort	5988	6659	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
74	random⍰	1500	quicksort	9264	10259	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
75	random⍰	2000	quicksort	12871	14199	y = 1.0021 (x) ^ 1.2534 w/ correlation 100.0 %	y = 1.0023 (x) ^ 1.268 w/ correlation 100.0 %
76	random⍰	50	quicksort_rand	132	162	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
77	random⍰	500	quicksort_rand	2507	2838	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
78	random⍰	1000	quicksort_rand	5445	6113	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
79	random⍰	1500	quicksort_rand	9843	10847	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
80	random⍰	2000	quicksort_rand	13870	15194	y = 0.9976 (x) ^ 1.252 w/ correlation 100.0 %	y = 0.9983 (x) ^ 1.2668 w/ correlation 100.0 %
81	random⍰	50	quicksort_mot	171	204	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
82	random⍰	500	quicksort_mot	2922	3266	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
83	random⍰	1000	quicksort_mot	5951	6629	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
84	random⍰	1500	quicksort_mot	9758	10748	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
85	random⍰	2000	quicksort_mot	14842	16182	y = 1.0004 (x) ^ 1.257 w/ correlation 100.0 %	y = 1.0009 (x) ^ 1.2713 w/ correlation 100.0 %
86	random⍰	50	natural_merge	270	244	y = 1.0018 (x) ^ 1.3211 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.3136 w/ correlation 100.0 %
87	random⍰	500	natural_merge	4256	4000	y = 1.0018 (x) ^ 1.3211 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.3136 w/ correlation 100.0 %
88	random⍰	1000	natural_merge	9512	9000	y = 1.0018 (x) ^ 1.3211 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.3136 w/ correlation 100.0 %
89	random⍰	1500	natural_merge	15244	14483	y = 1.0018 (x) ^ 1.3211 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.3136 w/ correlation 100.0 %
90	random⍰	2000	natural_merge	20997	19981	y = 1.0018 (x) ^ 1.3211 w/ correlation 100.0 %	y = 1.0016 (x) ^ 1.3136 w/ correlation 100.0 %
91	random⍰	50	2-way_merge	218	218	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %
92	random⍰	500	2-way_merge	3843	3843	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %
93	random⍰	1000	2-way_merge	8727	8727	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %
94	random⍰	1500	2-way_merge	13954	13954	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %
95	random⍰	2000	2-way_merge	19402	19402	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %	y = 1.0018 (x) ^ 1.3088 w/ correlation 100.0 %
96	random⍰	50	3-way_merge	207	156	y = 1.0017 (x) ^ 1.3007 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2531 w/ correlation 100.0 %
97	random⍰	500	3-way_merge	3637	2640	y = 1.0017 (x) ^ 1.3007 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2531 w/ correlation 100.0 %
98	random⍰	1000	3-way_merge	8236	5857	y = 1.0017 (x) ^ 1.3007 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2531 w/ correlation 100.0 %
99	random⍰	1500	3-way_merge	13167	9400	y = 1.0017 (x) ^ 1.3007 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2531 w/ correlation 100.0 %
100	random⍰	2000	3-way_merge	18327	13082	y = 1.0017 (x) ^ 1.3007 w/ correlation 100.0 %	y = 1.001 (x) ^ 1.2531 w/ correlation 100.0 %
101	random⍰	50	heap_sort	508	440	y = 1.0032 (x) ^ 1.4475 w/ correlation 100.0 %	y = 1.0029 (x) ^ 1.424 w/ correlation 100.0 %
102	random⍰	500	heap_sort	10214	8636	y = 1.0032 (x) ^ 1.4475 w/ correlation 100.0 %	y = 1.0029 (x) ^ 1.424 w/ correlation 100.0 %
103	random⍰	1000	heap_sort	23384	19784	y = 1.0032 (x) ^ 1.4475 w/ correlation 100.0 %	y = 1.0029 (x) ^ 1.424 w/ correlation 100.0 %
104	random⍰	1500	heap_sort	37584	31799	y = 1.0032 (x) ^ 1.4475 w/ correlation 100.0 %	y = 1.0029 (x) ^ 1.424 w/ correlation 100.0 %
105	random⍰	2000	heap_sort	52844	44612	y = 1.0032 (x) ^ 1.4475 w/ correlation 100.0 %	y = 1.0029 (x) ^ 1.424 w/ correlation 100.0 %