QR Sort: A New Non-Comparative Integer-Based Sorting Algorithm

Randolph Bushman[1*]

[1*]Military Resilient Systems, Johns Hopkins University APL, Johns Hopkins Road, Laurel, 20723, Maryland, United States.

Corresponding author(s). E-mail(s): randytbushman@gmail.com;

Abstract

In this paper, we introduce a new stable, non-comparative integer-based sorting algorithm with an attainable linear time complexity, $\mathcal{O}(n)$. This approach utilizes the Quotient Remainder Theorem in conjunction with Counting Sort subroutines. This algorithm divides each input array element by a selected positive integer divisor, d, to generate representative remainder and quotient keys. We perform Counting Sort on the remainder keys followed by the quotient keys. This method yields a time and space complexity of $\mathcal{O}(n + d + \frac{k}{d})$, where k represents the difference between maximum and minimum array values. If we set the divisor, d, equal to the length of the input array, n, the complexity simplifies to $\mathcal{O}(n + \frac{k}{n})$. As the length of the input array increases and the value of k remains constant, the time complexity converges to $\mathcal{O}(n)$. Our results indicate that our algorithm outperforms established integer-based sorting algorithms such as Counting Sort and Least Significant Digit Radix Sort with large input arrays. We present our methodology and comprehensive discussion of our results in the main body of the paper.

Keywords: Integer Sorting, Linear Sorting, Stable Sorting, Complexity Analysis

1 Introduction

Sorting refers to the procedure that arranges a sequence of array items in a specified order [1]. Computer scientists depend on efficient sorting algorithms

to perform tasks such as Binary Search, median finding, and prioritization tasks.

Comparison-based sorting algorithms sort array sequences with a single abstract comparison operator, often "less than or equal to" [2]. These algorithms possess a proven lower bound time complexity of $\mathcal{O}(n * log(n))$, where n represents the input array length. Insertion Sort, Selection Sort, Merge Sort, and Quicksort serve as classic examples of comparison-based sorting algorithms.

Non-comparative integer sorting algorithms, or integer-based sorting algorithms, classify a distinct group of sorting methods with no proven lower-bound time complexity. These algorithms arrange arrays of data values by integer keys and bypass the comparison-based sorting lower bound. The current theoretical optimal lower-bound time complexity to sort arbitrary integer array sequences stands at $\mathcal{O}(n)$.

This paper introduces QR Sort, a new integer-based sorting algorithm engineered to deliver faster performance than traditional integer-based methods. QR Sort divides each array element by a user-specified divisor and uses the acquired quotient and remainder values as sorting keys. Furthermore, QR Sort qualifies as a stable sorting algorithm which signifies that input array elements with equal keys maintain the initial relative order after sorting [4].

We organized the rest of the paper as follows: Algorithms, which discusses examples of integer-based sorting algorithms, Proof of Sorting and Stability, which proves the sorting and stability of QR Sort, Implementation and Optimizations, which discusses specific implementations of QR Sort, and Results and Discussion, which examines experimental data in depth.

2 Algorithms

2.1 Counting Sort

Counting Sort constitutes a stable integer-based sorting algorithm that counts the occurrences of each distinct key in the input array and stores them in a count array. Counting Sort then applies a cumulative sum on the count array such that the value at each index equals the sum of the previous values plus itself [5]. Next, it iterates through the input array in reverse order, refers to the corresponding index in the count array to find the final output array index, decrements that index value by one, and then places the array value in position in an auxiliary array.

The time and space complexity equals $\mathcal{O}(n + k)$, where k represents the difference between the maximum and minimum input array values. Figure 1 demonstrates how the cumulative count array determines the final index locations of the sorted elements.

With small relative k values to n, the time complexity of Counting Sort approaches $\mathcal{O}(n)$. However, the performance degrades when k grows, given that no correlation exists with n. In these cases, memory usage also increases as the length of the count array equals k. Despite the simple nature of this

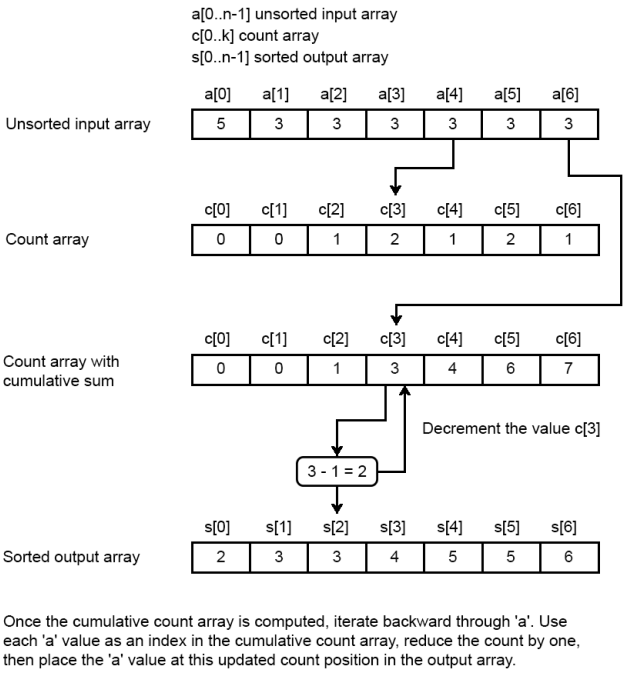

Fig. 1 Counting Sort utilizes the cumulative count array sums to determine the final index locations of the initial unsorted elements. Counting Sort first decrements the count and then uses that value to find the final index location.

algorithm, more sophisticated algorithms like Least Significant Digit Radix Sort and QR Sort utilize Counting Sort subroutines to mitigate performance losses caused by large k values.

2.2 Radix Sort

Least Significant Digit Radix Sort, or Radix Sort, constitutes another stable integer-based sorting algorithm. This algorithm sorts each input array element by the individual digits of each value from the least significant digit to the most significant digit. Radix Sort completes once it reaches the most significant digit of the largest value in the input array. Radix also operates on other positive number systems, which affects performance. Figure 2 illustrates how Radix Sort sorts an array of base-10 integers.

Let b represent the selected number system used to sort. Radix Sort invokes Counting Sort as a subroutine $floor(log_b(k))$ times as this represents the maximum number of digits in the largest input array value. As such, the time complexity of Radix Sort equals $\mathcal{O}((n+b) * log_b(k))$. This relationship reveals that the impact of k decreases exponentially as the length of the input array

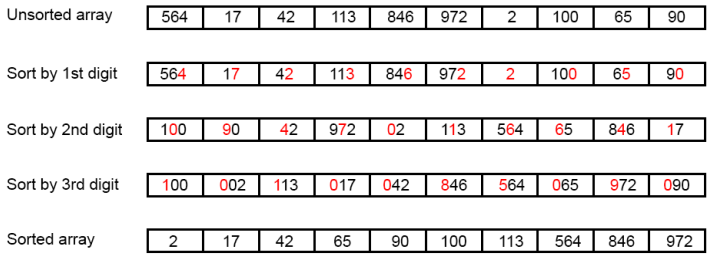

Fig. 2 Radix Sort instance that sorts base-10 integers.

increases. Therefore, as k increases, Radix Sort outperforms Counting Sort in time and space efficiency. In particular, base-n Radix Sort requires $\mathcal{O}(n)$ space instead of $\mathcal{O}(k)$.

2.3 QR Sort

QR Sort, like Radix Sort, leverages Counting Sort subroutines to sort array elements. Contrary to Radix Sort, which invokes $\mathcal{O}(log_b(k))$ Counting Sort subroutines, QR Sort invokes a maximum of two. QR Sort also necessitates a positive integer input, d, referred to as the divisor.

QR Sort divides each array element by the divisor d to generate quotient and remainder keys. It then invokes Counting Sort with the remainder keys first, followed by the quotient keys. QR Sort omits the quotient sort subroutine if the largest generated quotient equals zero. Figure 3 and Figure 4 illustrate the sorting process for the remainder and quotient keys.

The time and space complexity of QR Sort equals $\mathcal{O}(n + d + \frac{k}{d})$. If we set the divisor, d, equal to the array length, n, we simplify this complexity to $\mathcal{O}(n+\frac{k}{n})$. This refined time complexity suggests that the impact of k decreases hyperbolically as the length of the input array increases. The Results and Discussion section highlights scenarios where QR Sort outperforms Counting Sort and Radix Sort.

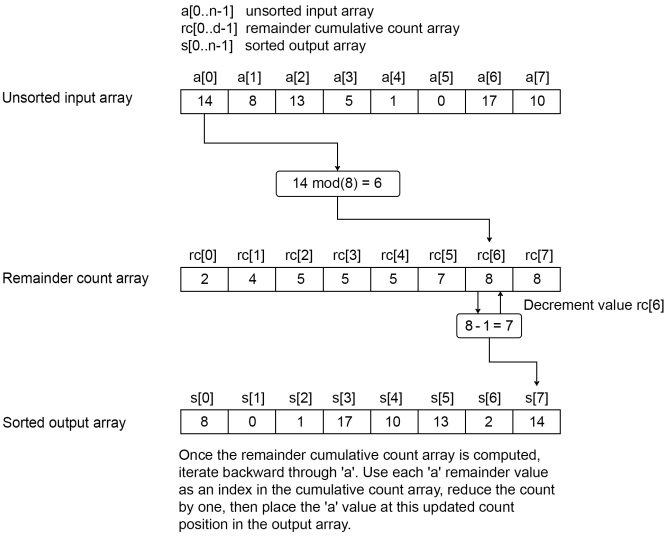

Fig. 3 Counting Sort used as a subroutine in QR Sort to sort the array elements by their remainders.

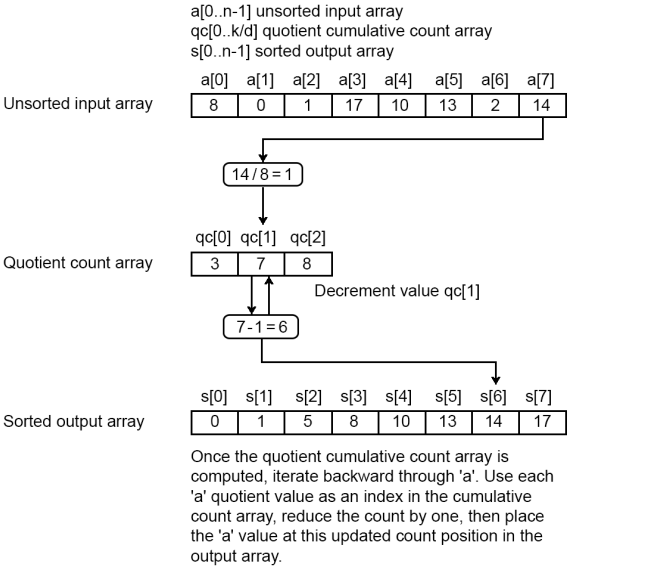

Fig. 4 Counting Sort used as a subroutine in QR Sort to sort the array elements by their quotients.

3 Proof of Sorting and Stability

Definition 1 Let x and y represent two arbitrary integer array elements. If x precedes y in the array, we write $x \longrightarrow y$. Let $z_r \equiv (z \mod d)$ and $z_q = floor(\frac{z}{d})$ for $x, d \in \mathbb{Z}^+$. The Quotient-Remainder Theorem states $z = dz_q + z_r$.

Lemma 3.1 *If $x < y$ and $x_r \geq y_r$ then $x_q < y_q$*

Proof We show that given the constraints, $x_q \leq y_q$.

$$x < y \implies \frac{x}{d} \leq \frac{y}{d} \implies x_q \leq y_q$$

We show that $x_q \neq y_q$ and conclude $x_q < y_q$. We assume $x_q = y_q$ and arrive at a contradiction.

$$x < y$$

We use the Quotient Remainder Theorem to produce the identity below.

$$x_r + dx_q < y_r + dy_q \implies x_r < y_r$$

The quotient terms cancel as we assumed $x_q = y_q$. Given the constraints of the problem, the above statement forms a false conclusion which implies $x_q \neq y_q$.

$$x_q \leq y_q \text{ and } x_q \neq y_q \implies x_q < y_q$$

□

Theorem 3.2 *QR Sort guarantees the sorting of array elements.*

Proof We assume $x < y$ then conclude that $x \longrightarrow y$ in the resulting sequence.

1. For $x < y$ and $x_r \geq y_r$

   By Lemma 1, $x_q < y_q$. This implies $x \longrightarrow y$ after the quotient sort executes.

2. For $x < y$ and $x_r < y_r$

   $x \longrightarrow y$ results after the remainder sort executes, since $x_r < y_r$. After the stable quotient sort executes, x and y maintain their relative positions since $x < y \implies x_q \leq y_q$.

□

Theorem 3.3 *QR Sort maintains the relative order of array elements with equal values and guarantees stability.*

Proof We let $x = y$ and conclude that if $x \longrightarrow y$ in the initial sequence, the result sequence produces the same relative order.

$$x = y \implies x_r = y_r$$

The remainder sort produces the same relative order of x and y.

$$x = y \implies x_q = y_q$$

The quotient sort produces the same relative order of x and y, which implies $x \longrightarrow y$ in the result sequence.

□

4 Implementation and Optimizations

4.1 QR Sort Implementation

In our QR Sort implementation, we employ two Counting Sort subroutines. We refer to the first subroutine as the 'remainder sort' and the second as the 'quotient sort'. For guidance on how to select the appropriate d, see the Divisor Selection section. To aid in our explanation, we use 'min-value' and 'max-value' to denote the minimum and maximum values in the array sequence. To better understand the process of sorting an array sequence with remainder keys, refer to Pseudo Code 1.

After we execute the remainder sort, we check if the maximum quotient value in the array equals zero. We calculate the maximum quotient value with the equation min-value $\frac{max\_value - min\_value}{d}$. If so, it allows us to bypass the quotient sort and end early. For a clearer understanding of the sorting process using quotient keys, refer to Pseudo Code 2.

4.2 Subtraction-Free Sorting

Earlier implementations of QR Sort, as shown in Pseudo Codes 1 and 2, subtracted the smallest input array value from each of the elements prior to key generation. This method enabled the sorting of negative integers and minimized the size of the count arrays. If we assume the minimum value equals zero, it allows QR Sort to exclude these subtractions and the linear search for the smallest array value.

This optimization presents notable drawbacks. First, it removes the ability to sort negative integers. Second, it decreases time and space efficiency in input arrays with large minimum values. The application of this approach best suits scenarios where the user holds intimate knowledge of the array to sort.

4.3 Divisor Selection - Powers of 2

While the length of the input array, n, stands as a convenient and effective default value for d, we note that no current universal optimal choice exists. There exists a countably infinite number of solutions for d, however, practical solutions lie between $1 < d < k$.

We leverage the use of powers of two, $d = 2^c$, to further increase efficiency, as this allows us to utilize the simple bitwise operations in binary arithmetic.

Algorithm 1 Sort Array with Remainder Keys Utilizing Counting Sort

```
1:  function REMAINDER-SORT(A, d)
2:      let n ⟵ A.length
3:      let j ⟵ 0
4:      let C[0..d − 1] be a new integer array
5:      let B[0..n − 1] be a auxiliary array for A
6:
7:      // Initialize count array.
8:      for i ← 0 to d − 1 do
9:          C[i] ← 0
10:     end for
11:
12:     // C[i] shall equal the remainder occurrence count of i in A.
13:     for i ← 0 to n − 1 do
14:         j ← (A[i] − min(A)) mod d
15:         C[j] ← C[j] + 1
16:     end for
17:
18:     // C[i] shall equal the cumulative remainder occurrence count of i in A.
19:     for i ← 1 to d − 1 do
20:         C[i] ← C[i] + C[i − 1]
21:     end for
22:
23:     // Use values in C to determine final output indexes.
24:     for i ← 0 down to n − 1 do
25:         j ← (A[i] − min(A)) mod d
26:         C[j] ← C[j] − 1
27:         B[C[j]] ← A[i]
28:     end for
29:
30:     // Copy B to A.
31:     for i ← 0 to n − 1 do
32:         A[i] ← B[i]
33:     end for
34:
35: end function
```

We compute the quotient and remainder values with the identities $\frac{n}{2^k} = n >> k$ and $n \pmod{2^k} = n \,\&\, (2^k - 1)$ where $>>$ represents the right bit-shift operation and & represents bitwise AND operation. This improves computational performance as modern processors incorporate hardware components designed for high-speed bitwise calculations.

Algorithm 2 Sort Array with Quotient Keys Utilizing Counting Sort

```
1:  function QUOTIENT-SORT(A, d)
2:      let n ⟵ A.length
3:      let j ⟵ 0
4:      let max_quotient ⟵ ((max(A) − min(A))/d) + 1
5:      let C[0..max_quotient] be a new integer array
6:      let B[0..n − 1] be a auxiliary array for A
7:
8:      // Initialize count array.
9:      for i ← 0 to max_quotient do
10:         C[i] ← 0
11:     end for
12:
13:     // C[i] shall equal the quotient occurrence count of i in A.
14:     for i ← 0 to n − 1 do
15:         j ← (A[i] − min(A))/d
16:         C[j] ← C[j] + 1
17:     end for
18:
19:     // C[i] shall equal the cumulative quotient occurrence count of i in A.
20:     for i ← 1 to max_quotient do
21:         C[i] ← C[i] + C[i − 1]
22:     end for
23:
24:     // Use values in C to determine final output indexes.
25:     for i ← n − 1 down to 0 do
26:         j ← (A[i] − min(A))/d
27:         C[j] ← C[j] − 1
28:         B[C[j]] ← A[i]
29:     end for
30:
31:     // Copy B to A.
32:     for i ← 0 to n − 1 do
33:         A[i] ← B[i]
34:     end for
35:
36: end function
```

4.4 QR Sort Generalization – Nested QR Sorts

In QR Sort, we recognize that the quotient sort becomes our bottleneck as k increases. To counter this, we invoke a recursive nested QR Sort on the quotient keys to replace the second Counting Sort subroutine. This works because QR Sort also constitutes a stable sorting algorithm, which falls in line with our proof. We shall either sort the quotients by a fixed number of nested QR Sorts

followed by a Counting Sort or an arbitrary number of nested QR Sorts until we reach the end-early condition. We meet the end-early condition when the maximum quotient equals zero, as discussed in the Implementation Discussion section.

We further generalize the time complexity to $\mathcal{O}((n+d)*m+\frac{k}{d^m})$, where m represents the number of nested QR Sort operations. For instance, if we perform two iterations of nested QR Sorts and initialize d to the array length, n, we obtain the time complexity $\mathcal{O}(n+\frac{k}{n^2})$.

We find that $\log_d(k)$ constitutes the largest value for m. If we equate m to this value, the time complexity develops into $\mathcal{O}((n+d)*\log_d(k)+\frac{k}{d^{\log_d(k)}})$ which simplifies to $\mathcal{O}((n+d)*log_d(k))$. This specific instance of the generalized QR Sort matches the time complexity of Radix Sort and performs identical operations at each iteration. We interpret each nested QR Sort as a Radix Sort at the mth digit in the base d number system.

The value for d need not remain constant at each iteration. Let D represent the ordered sequence of user-selected divisors used to sort at each iteration. We further generalize the time complexity of QR Sort.

$$\mathcal{O}\left(n\cdot|D|+\sum_{d\in D}d+k\prod_{d\in D}d^{-1}\right)$$

If we assume $1<d<k$ for each $d\in D$, we bound the sequence length with the inequality $1<|D|<\log_2(k)$. We require additional research and experimental data to determine if this method improves performance.

5 Results and Discussion

In this section, we explore and compare the performance of QR Sort against other algorithms mentioned earlier in this paper. We created SortTester_C, a C program that enables users to measure the performance of sorting algorithms [6]. SortTester_C allows users to specify array length ranges, randomly distributed array number ranges, and the number of repeated shuffled trials to conduct for each array length test. SortTester_C outputs comma-separated values (CSVs) with the average measured times for each sorting algorithm.

We conducted our experiments on a 2020 Razer Blade 15 Laptop with an Intel® Core™ i7-10750H CPU and 16GB of system memory. We compiled and executed the code on Windows 11, version 100.0.22621, build 22621 with the MinGW compiler with the maximum optimization flag, -O3, enabled.

5.1 General Algorithm Comparison

In our initial experiment, we compare the performance of Merge Sort, Quicksort, Counting Sort, Radix Sort, and QR Sort with the minimum value set to zero and the k equal to 50,000. We observe that the integer-based algorithms outperform the comparison-based algorithms by several orders of magnitude in Figure 5. As anticipated, because of the small relative k, Counting Sort emerged as the most efficient among the lot.

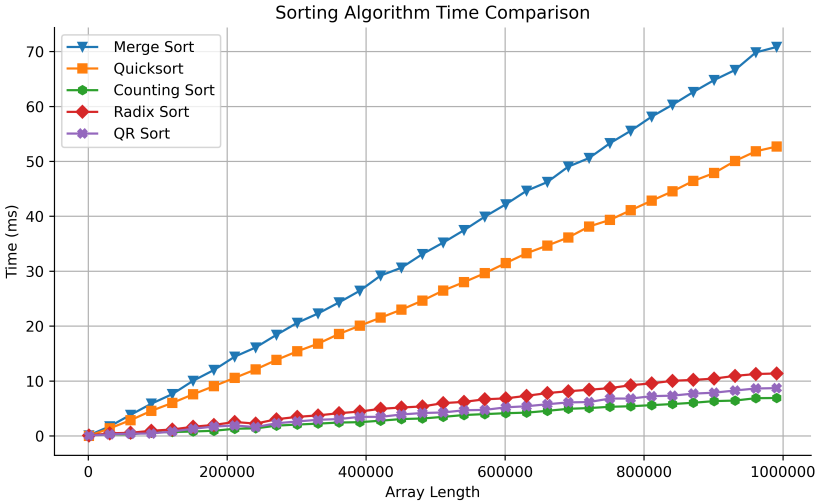

Fig. 5 Average time to sort arrays with Merge Sort, Quicksort, Counting Sort, Radix Sort, and QR Sort with $k = 50,000$.

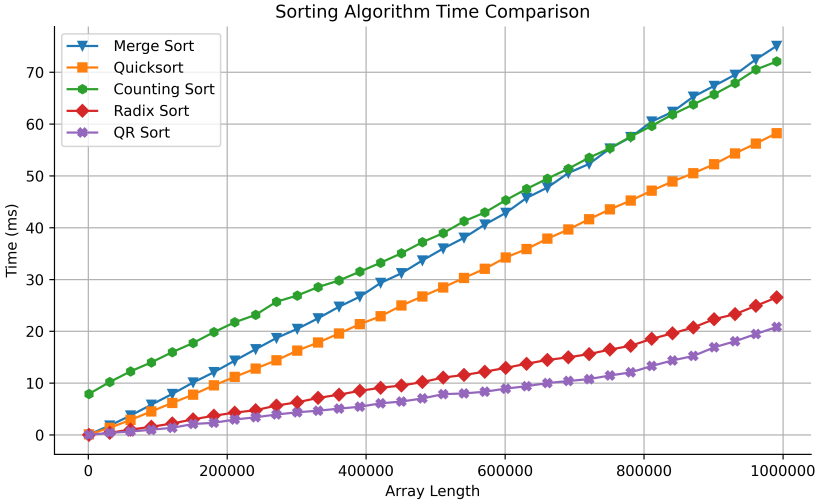

Fig. 6 Average time to sort arrays with Merge Sort, Quicksort, Counting Sort, Radix Sort, and QR Sort with $k = 5,000,000$.

We detailed in the Counting Sort subsection that while Counting Sort performs well with input arrays with small k values, the performance diminishes as k increases. We observe this trend in Figure 6 when we set k to 5,000,000. We also notice the increase in time taken to sort with QR Sort and Radix Sort while the performance of Merge Sort and Quicksort remains unchanged. We expect this given that both possess time complexities that exclude k.

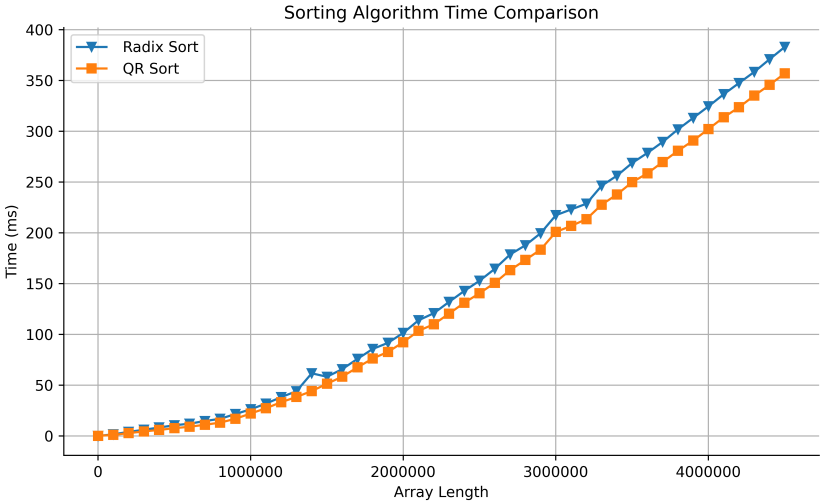

Fig. 7 Average time to sort arrays with Radix Sort and QR Sort with $k = 50,000,000$.

We further note an increase in slope between array lengths of 800,000 to 1,000,000 in the QR Sort and Radix Sort plots. We hypothesize this phenomenon stems from the additional arithmetic time needed to compute the remainders and quotients of the larger input array values. The observed slope from the Counting Sort plot remains constant, as this algorithm sidesteps these operations altogether.

5.2 Radix Sort Comparison

To further substantiate the superiority of QR Sort over Radix sort, we designed an experiment that extends the array lengths to 4.5 million and increased the k value to 50,000,000. Despite this, QR Sort continues to deliver superior performance, as shown in Figure 7.

This trend appears to persist as the array lengths increase. However, we identify an instance where QR sort fails by several orders of magnitude. We attribute the fast performance of QR Sort to the rapid convergence of the hyperbolic function term in the time complexity $\mathcal{O}(n + \frac{k}{n})$. Given that the hyperbola formed by the equation $\frac{1}{n}$ diverges to positive infinity as it approaches zero, we note a brief period where QR Sort underperforms with small array lengths and large k values, as depicted in Figure 8.

The time complexity of QR Sort hits an inflection point at $n = 2\sqrt{k}$, when $d = n$. We hypothesize that, for array lengths that exceed this value, the execution follows an almost linear trend.

5.3 QR Sort Optimization Comparison

In this section, we compare QR Sort to the optimization methods proposed earlier in the Implementation and Optimizations section and demonstrate how

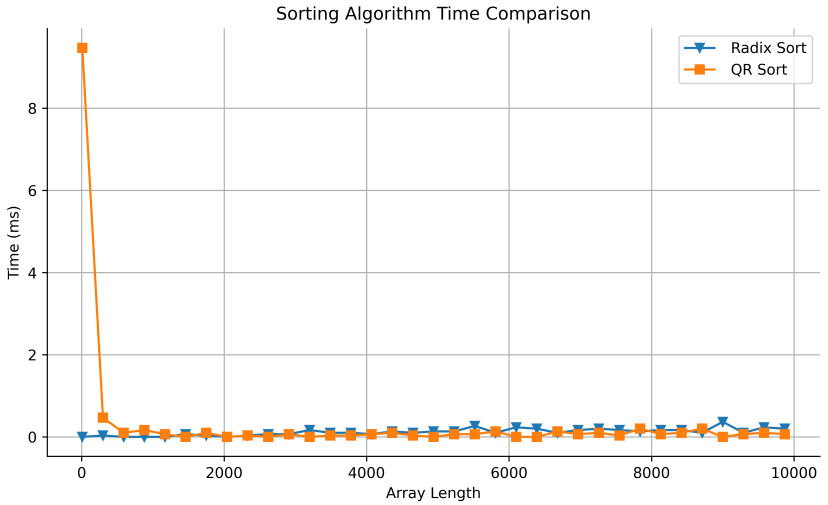

Fig. 8 Average time to sort arrays with Radix Sort and QR Sort with $k = 150,000,000$.

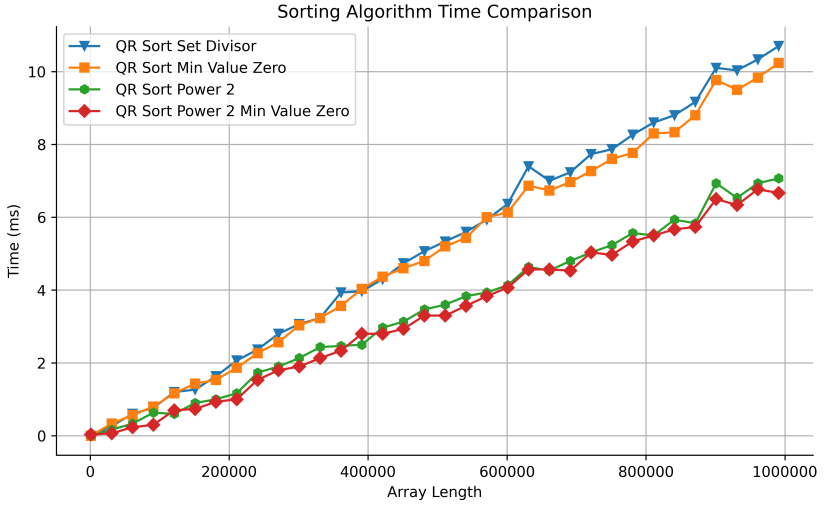

Fig. 9 Average time to sort arrays with QR Sort optimizations with $k = 1,000,000$ and $d = 256$.

each affects performance. For each experiment, we selected divisors that equal powers of two and populated the input arrays with values greater than or equal to zero.

The implementation of subtraction-free QR Sort and bitwise optimizations outperformed the other methods, as shown in Figure 9. As mentioned earlier, the subtraction-free implementation risks additional time and loses the ability to sort negative keys. The implementation with the bitwise optimizations alone

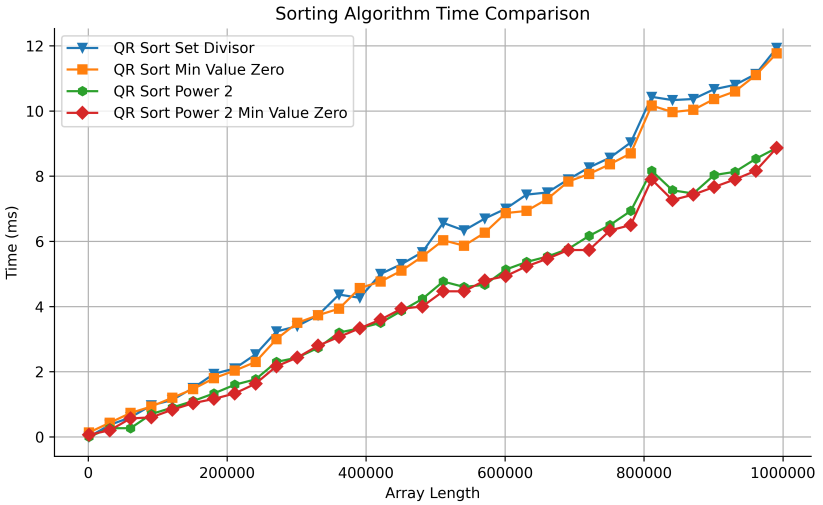

Fig. 10 Average time to sort arrays with QR Sort optimizations with $k = 1,000,000$ and $d = 65,536$.

yielded comparable performance and posed less risk to arbitrary input arrays. We show similar behavior in Figure 10 with a larger d value.

Each algorithm optimization performed close to two milliseconds slower than the previous experiment at $n = 1,000,000$. Despite this, the graph above exhibits similar trends to Figure 9.

6 Conclusion

Our experiments affirm the reliability of QR Sort as an integer-based sorting algorithm. QR Sort excels in tasks that involve large k values and surpasses traditional algorithms such as Counting Sort and Radix Sort. The convergence of QR Sort to linear time follows at a hyperbolic rate and even continues to outpace Radix Sort beyond the point of convergence. Our research highlighted the pivotal role of optimizations and divisor selection and paves the way for future research that explores optimal arguments for d and m in new iterations of QR Sort.

Declarations

- The author is employed by the Johns Hopkins University Applied Physics Laboratory, which holds ownership over the submitted work.
- No external funding was received to assist with the preparation of this manuscript.
- No external funding was received for conducting this study.
- No external funds, grants, or other support was received.

- While there are no other financial or non-financial interests to disclose, the author is affiliated with the Johns Hopkins University Applied Physics Laboratory, an organization that holds proprietary interests in the work produced by the author.
- While there are no competing interests to declare that are directly relevant to the content of this article, the author is an employee of the Johns Hopkins University Applied Physics Laboratory, which could have a financial or non-financial interest in the subject matter or materials discussed.
- The author declares that they are affiliated with the Johns Hopkins University Applied Physics Laboratory, an organization that may have a financial or non-financial interest in the subject matter or materials discussed in this manuscript.
- The author has no financial or proprietary interests in any material discussed in this article, apart from their employment by the Johns Hopkins University Applied Physics Laboratory, which holds proprietary interests in the author's work.

References

[1] K. Moore, et al., Sorting Algorithms, Brilliant, 2017. [Online; accessed 14-June-2022]. Available from: https://brilliant.org/wiki/sorting-algorithms/.

[2] M. Dey, "Time Complexity Bound for comparison based sorting," Opengenus, 2021. [Online; accessed 14-June-2022]. Available from: https://iq.opengenus.org/time-complexity-for-comparison-based-sorting/.

[3] P. J. Elder, "Linear Sorts," 2015. [Online; accessed 14-June-2022]. Available from: https://www.eecs.yorku.ca/course_archive/2015-16/F/2011/lectures/13%20Linear%20Sorts.pdf.

[4] D. Tang, "Introduction to Sorting Algorithm," cpp.edu, 2012. [Online; accessed 14-June-2022]. Available from: https://www.cpp.edu/~ftang/courses/CS241/notes/sorting.htm.

[5] K. Moore, "Counting Sort," Brilliant, 2017. [Online; accessed 14-June-2022]. Available from: https://brilliant.org/wiki/counting-sort/.

[6] R. Bushman, "SortingTester," GitHub. [Online; accessed 12-November-2022]. Available from: https://github.com/randytbushman/SortTester_C.