**Quotient Remainder Sort: An Integer-Based Sorting Method**

**Algorithm Report for Fall 2022**

**Randolph Bushman**

**ABSTRACT:** We present a new stable integer-based sorting algorithm that possesses an attainable linear time complexity, . We utilize the Quotient Remainder Theorem and Counting Sort subroutines to implement this sorting algorithm. We first select a positive integer, , which we shall refer to as the divisor. We then iterate through the input array and use the divisor to calculate the remainder and quotient values of each array element. The remainder and quotient values act as representative keys for each array element. We perform Counting Sort with the remainder keys and then the quotient keys to sort the array. This algorithm possesses the time and space complexity where represents the difference between the largest and smallest array values. This simplifies to if we initialize to equal the length of the array, . This algorithm converges to linear time as the length of the input array increases if the value remains constant. Even when grows, this algorithm outperforms classic integer-based algorithms like Counting Sort and LSD Radix Sort in the experiments conducted.

# Introduction

Sorting refers to the procedure that arranges a sequence of items in a specified order [1]. Computer scientists rely on efficient sorting algorithms for tasks such as Binary Search, median finding, and prioritization. Comparison-based algorithms represent a classification of sorting methods that sort array sequences with the use of a single abstract comparison operator, often “less than or equal to” [2]. Comparison­-based sorting algorithms possess the proven lower bound time complexity of ), where represents the input array length. Classic examples of comparison-based algorithms include Insertion Sort, Selection Sort, Merge Sort, and Quicksort.

Integer-based sorting algorithms refer to the separate classification of sorting methods that possess no proven lower-bound time complexity. Integer-based algorithms sort arrays of data values by integer keys instead of abstract comparison operations [3]. These algorithms offer significant potential for the development of sorting procedures as they tend to offer faster times, often pseudo-linear. Linear time or represents the theoretical optimal lower-bound time complexity required to sort arbitrary array sequences.

To deliver faster performance than traditional integer-based methods, we invented Quotient Remainder Sort (QR Sort). QR Sort divides each array element by a specified divisor and utilizes the quotient and remainder keys produced to sort the array. QR Sort also constitutes a stable sorting algorithm which implies the elements in the input array with equal keys maintain their relative order after the sort executes [4]. We organized this paper into the sections 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.

# Algorithms

## Counting Sort

Counting Sort constitutes a stable integer-based sorting algorithm. It first counts the number of elements that possess distinct key values. It then adds to each count, the sum of the counts with smaller key values. These values help determine the final index locations of the sorted elements in the output array [5]. Common implementations store these cumulative counts in integer arrays. Counting Sort possesses the time and space complexity of , where represents the difference between the largest and smallest values in the input array. Figure 1 shows how the cumulative count array determines the final index locations of the sorted elements.

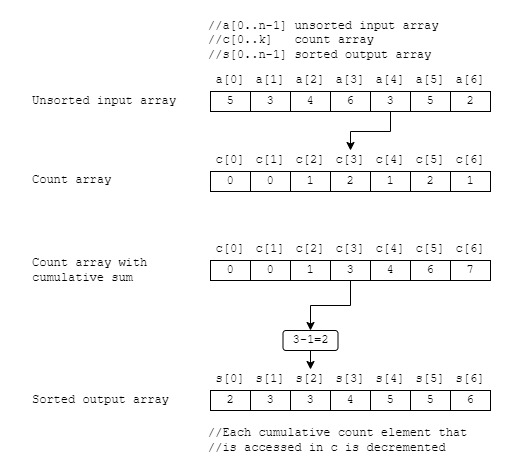


Figure : Counting Sort utilizes the cumulative count array sums to determine the final index locations of the initial unsorted elements.

Counting Sort converges to linear time as the length of the input array increases, and the value for remains constant. If the value for increases, Counting Sort underperforms. This occurs because and possess an independent relationship. Memory usage also increases since the length of the cumulative count array equals . While elementary, more sophisticated algorithms like LSD Radix Sort and QR Sort utilize Counting Sort in subroutines.

## Least Significant Digit (LSD) Radix Sort

LSD Radix Sort or Radix Sort represents another stable integer-based sorting algorithm. As the name implies, this algorithm sorts by the radixes, or digits, of the elements in the input array. Radix Sort uses the digits of the array elements as keys and invokes stable sorts in order from the LSD to the most significant digit (MSD). Once Radix Sort sorts by the MSD, it stops. Radix Sort also permits the use of other positive number systems which, in turn, alters the performance. Figure 2 illustrates how Radix Sort sorts a sequence of base 10 integers.

Table

Description automatically generated

Figure : Radix Sort instance that sorts base 10 integers.

Let represent the selected number system used to sort. Radix Sort invokes Counting Sort as a subroutine times as this represents the maximum number of digits in the largest input array element. Radix Sort, therefore, possesses the time complexity . This simplifies to if we initialize to equal the length of the input array, . This relationship indicates that the influence from decreases at an exponential rate as the length of the input array increases. As increases, Radix Sort outperforms Counting Sort in both time and space. Radix Sort requires space instead of .

## QR Sort

Like Radix Sort, QR Sort utilizes Counting Sort as a subroutine to sort array elements. While Radix Sort invokes Counting Sort subroutines, QR Sort invokes two. QR Sort also requires a positive integer input, , which we refer to as the divisor.

We divide each of the array elements by the divisor and acquire the quotient and remainder values. These values act as representative keys for each array element. We then invoke a stable sort on the elements in the input array with the remainder keys. We invoke another stable sort with the quotient keys. We utilize Counting Sort to sort the remainder and quotient sequences. Figure 3 and Figure 4 demonstrate how we sort the remainders and quotients in QR Sort.

A picture containing diagram

Description automatically generated

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

Diagram

Description automatically generated

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

QR Sort possesses the time and space complexity . This simplifies to if we initialize to equal the length of the input array, . This new time complexity shows that the influence from decreases at a hyperbolic rate as the length of the input array increases. In the Results and Discussion, we highlight situations where QR outperforms Counting Sort and Radix Sort.

# Proof of Sorting and Stability

## Notation and Background

Let and represent two arbitrary integer array elements. If precedes in the array, we write . Let and for . The Quotient-Remainder Theorem states that .

## Lemma 1. If and then .

*Proof.* We show that given the constraints, .

We show that to arrive at the conclusion We prove this with the statement and arrive at a contradiction.

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

The quotient terms cancel as we assumed . Given the constraints of the problem, the above statement forms a false conclusion which implies .

and ⇒

**QED**

## **Theorem 1.** QR-Sort guarantees the sorting of array elements.

*Proof.*We assume and conclude by exhaustion that in the result sequence.

1. For and

By Lemma 1,. This implies after the quotient sort executes.

1. For and

implies as shown in Lemma 1. After the remainder sort executes, since . After the stable quotient sort executes, and maintain their relative positions since .

**QED**

## Theorem 2. QR-Sort maintains the relative order of array elements with equal values and guarantees stability.

*Proof.*We assume and conclude that if in the initial sequence, the result sequence produces the same relative order.

⇒

The remainder sort produces the same relative order of and .

⇒

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

**QED**

# Implementation and Optimizations

## Implementation Discussion

We implement QR Sort with two Counting Sort subroutines. We refer to the first subroutine as the remainder sort and the second as the quotient sort. The parameters to QR Sort consist of the array to sort and the divisor. We discuss methods to select in the Divisor Selection section. Let *min\_value* and ­*max\_value* represent the smallest and largest values in the input array sequence. We present the pseudocode that sorts an array sequence in ascending order with remainder keys.

//Input: An array of integer elements and divisor

//Output: Array of the elements in in remainder ascending order

**for** **to** **do**

[i] ← 0

**for** **to** **do**

←

**for** **to** **do**

**for** **down to** **do**

**return** S

After we invoke the remainder sort, we execute a simple test that allows us to bypass the quotient sort and end early. We need not perform the quotient sort if the maximum quotient value in the input array equals a value less than or equal to one. We calculate the maximum quotient value with the equation . We present the pseudocode that sorts an array sequence in ascending order with quotient keys.

//Input: An array of remainder sorted integer elements and divisor used for //that remainder sort.

//Output: Array of the elements in in ascending order

**for** **to** **do**

[i] ← 0

**for** **to** **do**

←

**for** **to** **do**

**for** **down to** **do**

**return** S

## Minimum Value Zero

In the above implementations, we located the minimum input array value and subtracted it from each array element before the remainder and quotient sorts. This minimizes the size of the count array and enables us to sort negative integer values as well. If we assume the minimum value equals zero, we need not perform these subtractions. We also eliminate the need to search for the minimum array value. This optimization saves time with large array inputs when equals zero as shown in the Optimization Comparison section. With this optimization, however, we lose the ability to sort negative integers. In addition, we risk longer computation time and worse memory efficiency as shall depend on the maximum value alone.

## Divisor Selection – Powers of Two

While represents a reliable choice to initialize , there exists no proven universal optimal divisor. There exists an infinite number of possibilities for , however, practical divisor selections exist in the inequality . If we initialize to equal a power of two, it enables us to utilize bitwise operations to compute the remainder and quotient values. We compute the quotient and remainder values with the identities and where represents the right bit-shift operation and represents bitwise AND operation. In total, QR sort invokes division operations and modulo operations. This improves performance since modern processors possess hardware components that execute bitwise operations faster than regular quotient and remainder operations.

## QR Sort Generalization – Nested QR Sorts

In QR Sort, the Counting Sort subroutine used to perform the quotient sort develops into our bottleneck as increases. We need not rely on Counting Sort, however, to perform the quotient sort. We instead apply a nested QR Sort as we proved it constitutes a stable sorting algorithm. We either sort the quotients by a fixed number of nested QR Sorts followed by a Counting Sort on the quotient keys, 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 produces a value less than or equal to one as mentioned in the Implementation Discussion section. We generalize the time complexity to , where represents the number of iterations of nested QR Sorts. For instance, if we perform two iterations of nested QR Sorts, and initialize to the array length, , we obtain the time complexity .

We find that constitutes the largest value for . If we equate to this value, the time complexity develops into ) which simplifies to ). 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 th digit in the base number system.

The value for need not remain constant at each iteration, however. Let represent the ordered sequence of user-selected divisors used to sort at each iteration. This further generalizes the time complexity of QR Sort to . If we assume for each , we bound the sequence length with the inequality . We require additional research and experimental data to determine if this further generalized method improves performance.

# Results and Discussion

We show in the experiments that follow how the timed performance of QR Sort compares to the performance of the traditional algorithms mentioned earlier. For each graph, we conducted 30 trials and plotted the average completion times. For each trial, we generated copies of arrays with random permutations of linearly spaced values based on a given and measured the time for each algorithm to sort. We conducted the experiments on a 2020 Razer Blade 15 Laptop with an Intel® Core™ i7-10750H CPU and 16GB of system memory. We implemented the algorithms in Java and created a test driver program located in the SortTester GitHub repository [6]. We assume in our experiments that the Java garbage collector and JIT compiler shall not alter the proportions of the measured times at each trial.

## General Algorithm Comparison

We first compare the performance of QR Sort with to the traditional algorithms mentioned earlier in the paper. We observe the time taken to sort arrays with Merge Sort, Quicksort, Counting Sort, Radix Sort, and QR Sort in Figure 5.

Figure : Average time to sort arrays with Merge Sort, Quicksort, Counting Sort, Radix Sort, and QR Sort with .

The previous experiment shows how the integer-based algorithms outperformed the comparison-based algorithms. This occurred due to the small value relative to . We show in the experiment from Figure 6, how a larger value affects the performance of these algorithms.

Figure : Average time to sort arrays with Merge Sort, Quicksort, Counting Sort, Radix Sort, and QR Sort with .

As expected, the two comparison-based algorithms sorted the arrays at similar rates to the previous experiment in Figure 5. This occurred because Merge Sort and Quicksort possess time complexities independent of . Counting Sort, while faster than the comparison-based algorithms in the larger array trials, performed worse than in the previous experiment because it possesses a time complexity with a linear relationship between and . The time complexities QR Sort and Radix Sort decrease the influence of which explains the similar performance.

## Radix Sort Comparison

To demonstrate how affects the performance of QR Sort and Radix Sort, we test larger array lengths and ranges in the experiments that follow.

Figure : Average time to sort arrays with Radix Sort and QR Sort with .

The graph shows that QR Sort outperforms Radix Sort in most of the trials from Figure 7. The trend indicates a clear difference in performance between the two algorithms as the array lengths increase. To sort an array with a length of 100,000, Radix Sort requires almost twice the time compared to QR Sort. In Figure 8, we compare the performance after both algorithms converged to linear time when .

Figure : Average time to sort arrays with Radix Sort and QR Sort with .

We observe that when QR Sort outperforms Radix Sort. At this point, both algorithms converge to linear time. The trends from this graph also show how QR Sort maintains superiority in performance as array length increases. In Figure 9, we demonstrate how small relative to affect the results.

Figure : Average time to sort arrays with Radix Sort and QR Sort with .

We observe that Radix Sort outperforms QR Sort with smaller array lengths. The above trend at first exhibits a downward hyperbolic pattern which flattens over time. The quotient sort causes this as it takes time to calculate the cumulative sum of the count array.

## Optimization Comparison

We demonstrate in the experiments that follow how the optimized implementations of QR Sort affect performance. We generated arrays with values greater than or equal to zero and values that equate to powers of 2.

Figure : Comparison of QR Sort optimization algorithm performance with and .

The implementation of QR Sort with the minimum value of zero and bitwise optimizations outperformed the other methods as shown in Figure 10. As mentioned earlier, however, the minimum value zero implementation risks additional time and loses the ability to sort negative keys. The implementation with the bitwise optimizations alone yielded comparable performance and poses less risk to arbitrary input arrays. We show similar behavior in Figure 11 with a different value.

Figure : Comparison of QR Sort optimization algorithm performance with and .

Each of the algorithm implementations performed close to 4 seconds slower than the previous experiment. Despite this, however, the graph above exhibits similar trends to Figure 10.

## Conclusion

We conclude from our experiments that QR Sort constitutes a reliable integer-based sorting algorithm. In operations with large values, QR Sort outperforms classical algorithms like Counting Sort and Radix Sort. QR Sort converges to linear time at a hyperbolic rate and continues to outperform Radix Sort after the point of convergence. In addition, we demonstrated how the optimizations and divisor selection help improve computational performance as well. The conducted work presents the opportunity for additional research to prove or disprove the existence of the optimal arguments for and in QR Sort to further improve performance.

# References

|  |  |
| --- | --- |
| [1] | G. P. K. T. J. K. T. K. Karleigh Moore, "Sorting Algorithms," Brilliant, 22 September 2017. [Online]. Available: https://brilliant.org/wiki/sorting-algorithms/. [Accessed 14 June 2022]. |
| [2] | M. Dey, "Time Complexity Bound for comparison based sorting," Opengenus, 2 8 2021. [Online]. Available: https://iq.opengenus.org/time-complexity-for-comparison-based-sorting/. [Accessed 14 June 2022]. |
| [3] | P. J. Elder, "Linear Sorts," 1 December 2015. [Online]. Available: https://www.eecs.yorku.ca/course\_archive/2015-16/F/2011/lectures/13%20Linear%20Sorts.pdf. [Accessed 14 June 2022]. |
| [4] | D. Tang, "Introduction to Sorting Algorithm," cpp.edu, November 2012. [Online]. Available: https://www.cpp.edu/~ftang/courses/CS241/notes/sorting.htm. [Accessed 14 June 2022]. |
| [5] | K. Moore, "Counting Sort," Brilliant, 22 September 2017. [Online]. Available: https://brilliant.org/wiki/counting-sort/. [Accessed 14 June 2022]. |
| [6] | R. Bushman, "SortingTester," GitHub, [Online]. Available: https://github.com/randytbushman/SortingTester. [Accessed 19 May 2022]. |