**QR Sort: An Integer-Based Sorting Method**

**Algorithm Report for Fall 2022**

**Randolph Bushman**

**ABSTRACT:** We present a new stable integer-based sorting algorithm with 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. Next, we iterate through the input array and divide by each array element to compute the remainder and quotient values as they act as representative keys in the subroutines. We perform Counting Sort on the input array values with the remainder keys followed by the quotient keys to sort the array. This algorithm possesses the time and space complexity where represents the difference between the maximum and minimum array values. The time and space complexity 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 when the value remains constant. Even when grows, this algorithm outperforms classic integer-based algorithms like Counting Sort and LSD Radix Sort, as shown in the Results and Discussion section.

# 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 sorting algorithms sort array sequences with a single abstract comparison operator, often "less than or equal to" [2]. Comparison­-based sorting algorithms also 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 with no proven lower-bound time complexity. Integer-based sorting algorithms sort arrays of data values by integer keys instead of abstract comparison operations [3]. Integer-based sorting methods often execute faster than comparison-based algorithms as they bypass the comparison-based lower bound. Linear time or represents the theoretical optimal lower-bound time complexity required to sort arbitrary array sequences with integer keys.

We invented QR Sort, an integer-based sorting algorithm, to deliver faster performance than traditional integer-based methods. QR Sort divides each array element by a user-specified divisor and utilizes the acquired quotient and remainder values as keys to sort. QR Sort also constitutes a stable sorting algorithm which implies the elements in the input array with equal keys maintain their prior relative order after the sort executes [4]. We organized this paper into the sections Algorithms which discusses examples of integer-based sorting algorithms, which prove 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 counts the number of elements that possess distinct key values and then performs a cumulative sum on those counts. The cumulative counts help determine the final index locations of the sorted elements in the output array [5]. Standard Counting Sort implementations store the cumulative counts in integer arrays. Counting Sort possesses the time and space complexity of , where represents the difference between the maximum and minimum input array values. Figure 1 shows how the cumulative count array determines the final index locations of the sorted elements.

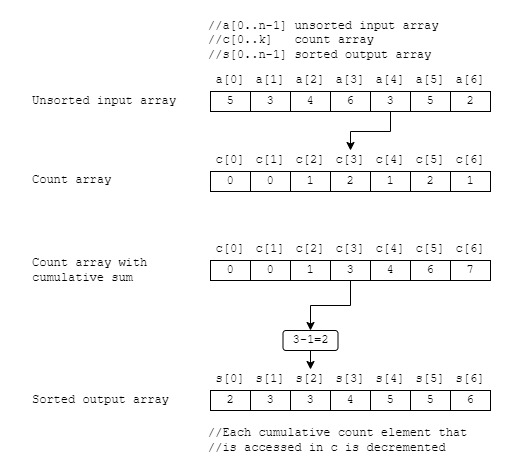


Figure : Counting Sort utilizes the cumulative count array sums to determine the final index locations of the initial unsorted elements.

Counting Sort converges to linear time as the length of the input array increases when the value for remains constant. The performance of Counting Sort decreases as increases since 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 to increase performance on input arrays with large values.

## Least Significant Digit (LSD) Radix Sort

LSD Radix Sort or Radix Sort represents another stable integer-based sorting algorithm that sorts by the 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 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 of the maximum input array value. Radix Sort, therefore, possesses the time complexity . This time complexity simplifies to if we initialize to equal the input array length, . 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. Base- 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.

QR Sort divides the array values by the to acquire quotient and remainder keys. Next, QR Sort invokes a stable sort with the remainder keys and another with the quotient keys. This implementation utilizes Counting Sort to sort the remainder and quotient sequences. Figure 3 and Figure 4 demonstrate how we sort the remainder and quotient keys.

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 input array length, . This new time complexity shows that the influence from decreases at a hyperbolic rate as the length of the input array increases. The Results and Discussion 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 values. 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 and , 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 of 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 minimum and maximum 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**

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**

## 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. We subtract this value from each element accessed from the input array to minimize the size of the count array and 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. 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 between 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 optimization 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. However, we need not rely on Counting Sort 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 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. We further generalize 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 method improves performance.

# Results and Discussion

In this section, we conducted several experiments that visualize and compare the performance of QR Sort to the algorithms mentioned in the previous sections. 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 the 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.

## General Algorithm Comparison

In the first 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 equal to 50,000. We observe that the integer-based algorithms outperform the comparison-based algorithms by several orders of magnitude in Figure 1. As expected, Counting Sort outperforms the competition.

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

We explained earlier in the Counting Sort subsection that while Counting Sort performs well with input arrays with small values, the performance decreases as increases. We obeserve this trend in Figure 6 when we set equal to 5,000,000. We also examine 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 since both possess time complexities absent of .

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

We also observe the increased slope from array length 800,000 to 1,000,000 in the QR Sort and Radix Sort plots. We hypothesize this phenomenon originates from the longer arithmetic times necessary to compute the remainders and quotients of the larger input array values. The observed slope from the Counting Sort plot remained constant since it avoids these operations altogether.

## Radix Sort Comparison

To demonstrate that QR Sort maintains faster performance than Radix Sort, we created an experiment that pushed the array lengths to 4.5 million and increased the value to 50,000,000. However, even with these extreme conditions, QR Sort yields superior results, as shown in Figure 7.

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

While this trend seems to continue as the array lengths increase, we observe a case when QR Sort fails by several orders of magnitude. The fast performance of QR Sort stems from the rapid convergence of the hyperbolic function term in the time complexity ). Since the hyperbola formed by the equation also diverges to positive infinity near the origin, array length zero, we observe a brief period when QR Sort underperforms with small array lengths and large values in Figure 8.

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

The time complexity of QR Sort reaches the inflection point when when . We hypothesize that array lengths greater than this value shall execute with virtually linear trends.

## Optimization Comparison

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

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

The implementation of QR Sort with the minimum value of zero and bitwise optimizations outperformed the other methods, as shown in Figure 9. As mentioned earlier, 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 posed less risk to arbitrary input arrays. We show similar behavior in Figure 10 with a larger value.

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

Each algorithm implementation performed close to two milliseconds slower than the previous experiment at . Despite this, the graph above exhibits similar trends to Figure 9.

## 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. Furthermore, 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 improve performance further.

# 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/SortTester\_C. [Accessed 12 November 2022]. |