Contents

Team Members:

Shivam Pandya

Abhishek Singh

Radix & Quick Sort Comparison

1. Introduction

This document presents the results of a performance comparison between Radix Sort (both RadixSort1 and RadixSort2 implementations) and Quick Sort, focusing on sorting arrays of different sizes and digit lengths. The testing involved arrays with sizes of 1,048,576 and 4,194,304 elements, and the sorting algorithms were evaluated on both 5-digit and 10-digit numbers.

2. Sorting Algorithms

1. Radix Sort (RadixSort1)
    - **Description:** RadixSort1 is a variant of Radix Sort where sorting is performed digit by digit, using different radix (base) values. The implementation uses a stable counting sort as a subroutine.
    - **Parameters Tested:** Number of digits (5 and 10), radix values (1, 3, 5).
2. Radix Sort (RadixSort2)
    - **Description:** RadixSort2 is another variant of Radix Sort that processes digits differently, potentially using more optimized techniques.
    - **Parameters Tested:** Number of digits (10 only).
3. Quick Sort
    - **Description:** Quick Sort is a comparison-based sorting algorithm that uses divide-and-conquer to partition the array and sort the partitions recursively.
    - **Parameters Tested:** Number of digits (5 and 10).

3. Test Results

Results from Computer 1: Dell Windows

Algorithm	Digits	List	Array Size - 1,048,576 (ms)	Array Size - 4,194,304 (ms)
RadixSort1 (r=1)	5	1	103	422
RadixSort1 (r=1)	5	2	133	415
RadixSort1 (r=1)	5	3	171	327
RadixSort1 (r=3)	5	1	281	563
RadixSort1 (r=3)	5	2	112	479
RadixSort1 (r=3)	5	3	130	199
RadixSort1 (r=5)	5	1	163	460
RadixSort1 (r=5)	5	2	70	167
RadixSort1 (r=5)	5	3	112	462
RadixSort1 (r=1)	10	1	273	1,000

RadixSort1 (r=1)	10	2	242	1,283
RadixSort1 (r=1)	10	3	251	905
RadixSort1 (r=3)	10	1	376	915
RadixSort1 (r=3)	10	2	327	712
RadixSort1 (r=3)	10	3	230	676
RadixSort1 (r=5)	10	1	325	1,085
RadixSort1 (r=5)	10	2	268	873
RadixSort1 (r=5)	10	3	163	865
RadixSort2	5	1	43	87
RadixSort2	5	2	61	201
RadixSort2	5	3	49	152
RadixSort2	10	1	38	165
RadixSort2	10	2	48	211
RadixSort2	10	3	52	115
QuickSort	5	1	100	566
QuickSort	5	2	91	553
QuickSort	5	3	91	594
QuickSort	10	1	98	467
QuickSort	10	2	96	435
QuickSort	10	3	97	441

Results from Computer 2: M2 Macbook

Algorithm	Digits	List	Array Size - 1,048,576 (ms)	Array Size - 4,194,304 (ms)
RadixSort1 (r=1)	5	1	28	79
RadixSort1 (r=1)	5	2	22	85
RadixSort1 (r=1)	5	3	20	77
RadixSort1 (r=3)	5	1	19	77
RadixSort1 (r=3)	5	2	19	76
RadixSort1 (r=3)	5	3	20	78
RadixSort1 (r=5)	5	1	19	76
RadixSort1 (r=5)	5	2	19	75
RadixSort1 (r=5)	5	3	19	75
RadixSort1 (r=1)	10	1	47	189

RadixSort1 (r=1)	10	2	66	187
RadixSort1 (r=1)	10	3	46	185
RadixSort1 (r=3)	10	1	46	185
RadixSort1 (r=3)	10	2	46	186
RadixSort1 (r=3)	10	3	46	185
RadixSort1 (r=5)	10	1	46	186
RadixSort1 (r=5)	10	2	46	186
RadixSort1 (r=5)	10	3	46	207
RadixSort2	5	1	19	46
RadixSort2	5	2	12	46
RadixSort2	5	3	11	60
RadixSort2	10	1	6	24
RadixSort2	10	2	5	26
RadixSort2	10	3	6	25
QuickSort	5	1	60	255
QuickSort	5	2	55	250
QuickSort	5	3	55	247
QuickSort	10	1	60	258
QuickSort	10	2	60	260
QuickSort	10	3	61	263

4. Observations and Analysis

1. Performance of RadixSort1:
    - For 5-digit numbers, RadixSort1 generally performs well with radix values of 1 and 3, but radix 5 shows significant improvements in some cases, especially for larger arrays.
    - For 10-digit numbers, RadixSort1 performance varies significantly with radix values. Higher radix values tend to increase the sorting time, which may be due to increased complexity in the digit processing.
2. Performance of RadixSort2:
    - RadixSort2 consistently outperforms RadixSort1 in terms of time for 10-digit numbers across all test cases. This indicates that RadixSort2 might be more optimized or have more efficient digit processing.
    - RadixSort2 was not tested with 5-digit numbers because its design may not efficiently handle smaller digit lengths or might be optimized specifically for longer digit lengths.
3. Performance of Quicksort:
    - Quicksort shows relatively consistent performance across both digit lengths but is generally slower compared to RadixSort2.
    - For larger arrays, Quicksort takes more time, reflecting its comparative inefficiency in handling large datasets compared to Radix Sort variants.

5. Conclusion

- **RadixSort1:** Shows variability in performance with different radix values and digit lengths. Optimal performance is achieved with lower radix values for 5-digit numbers and with specific conditions for 10-digit numbers.
- **RadixSort2:** Demonstrates superior performance for 10-digit numbers compared to RadixSort1 and Quicksort, making it a preferable choice for larger datasets.
- **Quicksort:** While consistent, it is generally slower compared to RadixSort2, especially for large arrays and higher digit lengths.

6. Recommendations

- For sorting large datasets with 10-digit numbers, RadixSort2 is recommended due to its superior performance.
- For 5-digit numbers, RadixSort1 with radix 5 performs well, but it's beneficial to test with different radix values to find the optimal configuration.
- Quicksort, while reliable, may not be the most efficient choice for vast arrays compared to Radix Sort algorithms.

7. Challenges Faced During Implementation and Testing

1. Implementation of RadixSort2:
    - Implementing RadixSort2 was particularly challenging due to limited experience with bit manipulation techniques. The complexity of bitwise operations and their integration into the sorting process required a significant learning curve and adaptation.
2. Performance Measurement Accuracy:
    - Accurate measurement of execution time and ensuring consistency in test conditions were crucial. Variations in system load and environment could affect results, necessitating multiple test runs and result averaging.
3. Complexity in Comparison:
    - Comparing performance across different algorithms, digit lengths, and array sizes required meticulous analysis to ensure valid and reliable results. Ensuring that the comparison was fair and comprehensive was a significant challenge.