Formulation and Analysis of MSD Radix Sort Algorithm with Other Algorithms

Abstract

We present a unified treatment of a number of related in-place MSD radix sort algorithms with varying algorithms. These algorithms use the idea of in-place partitioning which is a considerable improvement over the traditional linked list implementation of radix sort that uses O(n) space. We present formulation and analysis of MSD radix sort algorithm and other four algorithms, which is Timsort, Dual-pivot Quicksort, Huskysort, and LSD radix sort. We use Benchmark to compare the performance of the five methods. Results are shown that MSD radix sort has the worst performance while LSD radix sort has the shortest sorting time and the best performance.

1.Introduction

Sorting is a fundamental problem in computer science, with wide applications [1]. The moment the algorithm run-time and space complexity were formalized, sorting algorithms were at the forefront for analysis and improvement. Early recognition was given to Quicksort [2] and Heapsort [3] algorithms, because they improve on the O(n2) order of running time of other slow sorting algorithms, to O(n log n). Heapsort is O(n log n) in the worst case, while Quicksort is O(n log n) in the average case. In practice, Quicksort runs faster than Heapsort most of the time, although it exhibits O(n2) worst-case order of running time, which happens, for example, when the input data is nearly sorted. In general, sorting algorithms can be divided into two categories: ‘comparison-based’ and ‘distribution-based’. A comparison based algorithm, like Heapsort or Quicksort, sorts by comparing two elements at a time. On the other hand, a distribution-based algorithm, like radix sort [4–7], works by distributing the elements into different piles based on their values. Radix sort algorithms fall into two classes: MSD (most significant digit) and LSD (least significant digit). Radix sort algorithms process the elements in stages, one digit at a time. A digit is a group of consecutive bits with the digit size (number of bits) set at the beginning of the algorithm. MSD radix sort starts with the most significant (leftmost) digit and moves toward the least significant digit. LSD radix sort does it the other way. LSD distributes the elements into different groups – commonly known as ‘buckets’ and treated as queues (first-in-first-out data structure) – according to the value of the least significant (rightmost) digit. Then the elements are re-collected from the buckets and the process continues with the next digit. On the other hand, MSD radix sort first distributes the elements according to their leftmost digit and then calls the algorithm recursively on each group. MSD needs only to scan distinguishing prefixes, while all digits are scanned in LSD. For example, for radix-2 MSD (i.e. digit size = 1 bit), two buckets are used and the elements are distributed into either bucket depending on the value of the most significant bit. Then the process continues with the next bit, considering only groups that have more than one element, until all the bits have been scanned. Clearly, such an algorithm uses O(n) space for the combined two buckets and is able to sort n non-negative integers in the range [0,m] in O(n log m) order of running time.

2.Method

`

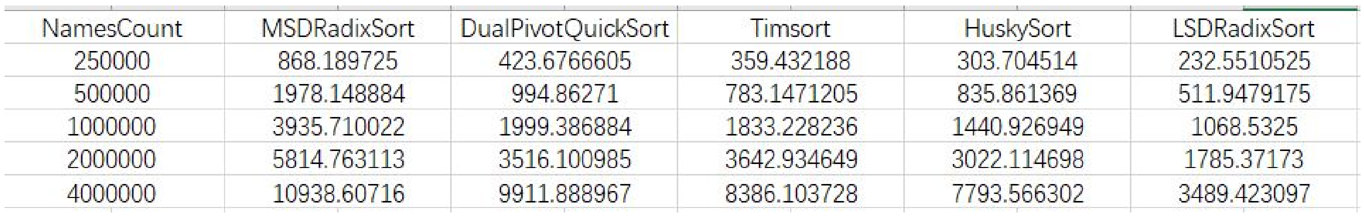
```
/**
 * Sort from a[lo] to a[hi] (exclusive), ignoring the first d characters of each String.
 * This method is recursive.
 *
 * @param a  the array to be sorted.
 * @param lo the low index.
 * @param hi the high index (one above the highest actually processed).
 * @param d  the number of characters in each String to be skipped.
 */
private static void sort(String[] a, int lo, int hi, int d) {
    if (hi < lo + cutoff) InsertionSortMSD.sort(a, lo, hi, d);
    else {
        int[] count = new int[radix + 2];        // Compute frequency counts.
        for (int i = lo; i < hi; i++) {
            count[sortAt(a[i], d) + 2]++;
        }
        for (int r = 0; r < radix + 1; r++)      // Transform counts to indices.
            count[r + 1] += count[r];
        for (int i = lo; i < hi; i++)     // Distribute.
            aux[count[sortAt(a[i], d) + 1]++] = a[i];
        // Copy back.
        if (hi - lo >= 0) System.arraycopy(aux, 0, a, lo, hi - lo);
        // Recursively sort for each character value.
        for (int r = 0; r < radix; r++)
            sort(a, lo + count[r], lo + count[r + 1], d + 1);
    }
}

private static int sortAt(String s, int d) {
    if (d < s.length()) {
        byte[] bytes = collator.getCollationKey(String.valueOf(s.charAt(d))).toByteArray();
        if (bytes.length < 7) {
            return (bytes[0] & 255) * 255;
        } else
            return (bytes[0] & 255) * 255 + (bytes[1] & 255);
    } else return -1;
}
```

`

3.Experiment

In this paper, a benchmark timer class is used to test the performance of five algorithms.



NamesCount	MSDRadixSort	DualPivotQuickSort	Timsort	HuskySort	LSDRadixSort
250000	868.189725	423.6766605	359.432188	303.704514	232.5510525
500000	1978.148884	994.86271	783.1471205	835.861369	511.9479175
1000000	3935.710022	1999.386884	1833.228236	1440.926949	1068.5325
2000000	5814.763113	3516.100985	3642.934649	3022.114698	1785.37173
4000000	10938.60716	9911.888967	8386.103728	7793.566302	3489.423097

Figure 1 Results of five methods

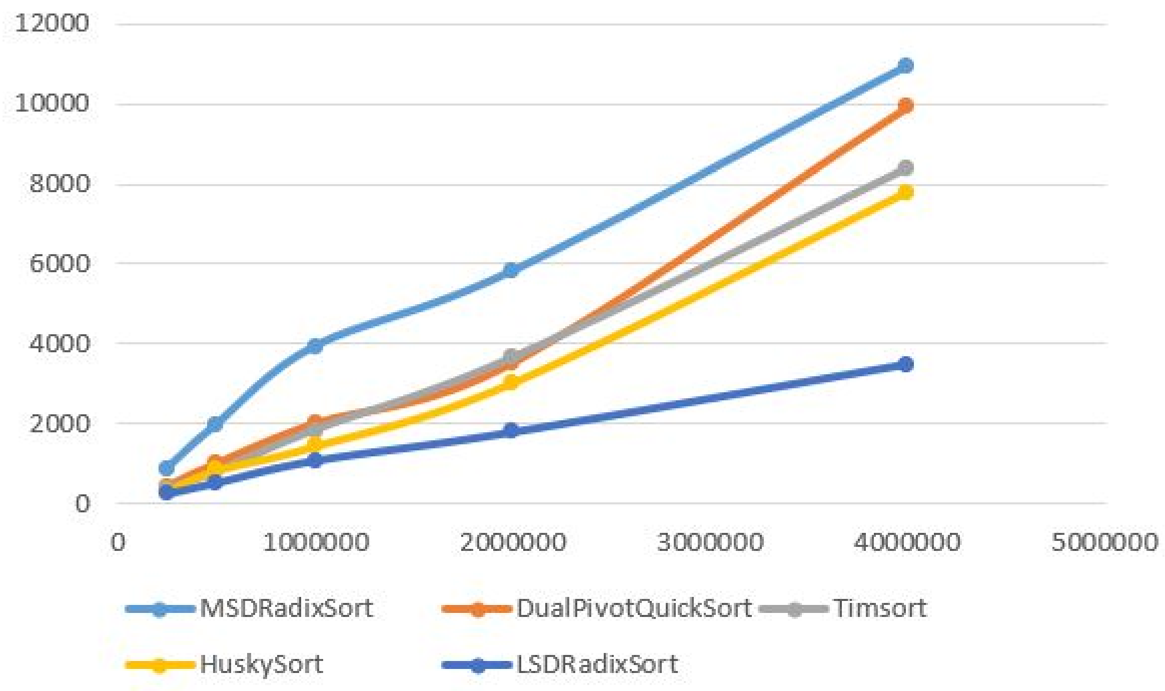


Figure 2 Performance of five methods

In this paper, five Chinese data sets are implemented to compare the performance of five algorithm methods. The test results can be clearly shown in the previous figure.

4.Conclusions

From the experiment result, we can get running time of the algorithms: LSDRadixSort < HuskySort < TimSort < DualPivotQuickSort < MSDRadixSort The sorting time of five algorithms increases almost linearly with the increase of array length. MSD Radix Sort has the longest sorting time and the worst performance. LSD Radix Sort has the shortest sorting time and the best performance.

References

- [1] D.E. Knuth, The Art of Computer Programming – Vol. 3 Sorting and Searching (Addison-Wesley, Reading, MA, 1973).
- [2] C.A.R. Hoare, Quicksort, Computer Journal 5(1) (1962) 10–16.
- [3] J.W. Williams, Algorithm 232: Heapsort, Communications of the ACM 7(6) (1964) 347–8.
- [4] P. Hildebrandt and H. Isbitz, Radix exchange – an internal sorting method for digital computers, Journal of the ACM 6(2) (1959) 156–63.
- [5] I.J. Davis, A fast radix sort, The Computer Journal 35(6) (1991) 636–42.
- [6] R. Sedgewick, Algorithms (Addison-Wesley, Reading, MA, 1988).
- [7] P.M. McIlroy, K. Bostic and M.D. McIlroy, Engineering radix sort, Computing Systems 6(1) (1993) 5–27.