Implementation of MSD Radix Sort for Natural Language

Ankur Chauhan, Aparna Anil Shelar, Mohit Nagpal

Abstract: An implementation of MSD Radix sort for natural language which uses Unicode characters in Hindi (Devanagari) and Chinese (Pinyin) languages. Performance analysis done in compared to other sorting algorithmic techniques such as LSD Radix Sort, Tim Sort, Quick Dual Pivot and Husky Sort and represented using graphs. A literature survey is done from related Research Papers describing MSD Radix Sort techniques in detail.

1. Introduction

In computer science, radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. MSD radix sorts are most suitable for sorting strings or fixed-length integer representations. A sequence like [b, c, e, d, f, g, ba] would be sorted as [b, ba, c, d, e, f, g]. If lexicographic ordering is used to sort variable-length integers in base 10, then numbers from 1 to 10 would be output as [1, 10, 2, 3, 4, 5, 6, 7, 8, 9], as if the shorter keys were left-justified and padded on the right with blank characters to make the shorter keys as long as the longest key. MSD sorts are not necessarily stable if the original ordering of duplicate keys must always be maintained. Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence. Having said that radix sort is not limited to integers. Because integers can be used to represent strings by hashing the strings to integers, radix sort works for data types other than integer as well.

The algorithm is named radix sort as it specifies the radix r to be used which changes how the sort is performed. The radix, or base, of the number system is the number of digits that represent a single position in the number; a radix of 2 is binary (0-1), 10 is decimal (0-9), 16 is hexadecimal (0-F) and so on. Since the radix determines the number of buckets in addition to the word size w used in the algorithm, changing it can drastically change how the sort plays out.

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. LSD radix sort performs a counting sort on the provided list for each digit, starting from the least significant digit. LSD radix sort is stable, unlike the MSD variant as the relative order is retained after each sorting iteration. MSD radix sort performs a counting sort on the provided list for each digit, starting from the most significant digit. The biggest problem with LSD radix sort is that it starts at the digits that make the least difference. If we could start with the most significant digits, the first pass would go a long way toward sorting the entire range, and each pass after that would simply handle the details. The idea of MSD radix sorting is to divide all of the digits with an equal value into their own bucket, then do the same thing with all of the buckets until the array is sorted. Naturally, this suggests a recursive algorithm, but it also means that we can now sort variable length items and we don't have to touch all of the digits to get a sorted array. This makes MSD radix sort considerably faster and more useful.

2. Literature survey

This survey is done on the Formulation and Analysis of in-place MSD radix sort algorithms paper.

Agenda: The author wanted to implement in place partition of Radix Sort over the traditional linked list implementation which uses O(n) space. He worked on MSD Radix sort with various radices collectively referred to as Matesort algorithms. The author presented formulation and analysis for three different approaches sequential, divide and conquer and permutation loop for partitioning by the general radix Matesort algorithm.

Binary Matesort Algorithm: The binary Matesort algorithm has some striking resemblance to Quicksort – the difference is in the extra 'bitloc' (bit location) input parameter and the partition method. Partitioning and Bit Partition Problem are the issues with Binary Matesort Algorithm. The issues can be avoided by considering different pivot in induction step. Quicksort is known to suffer from two problems that cause performance degradation – see Table 2 – whilst Matesort seems to be free of these problems. The first problem is that the algorithm (i.e. repeated partitioning) is slow – in comparison with other sorting algorithms – when applied to small size data, especially for data that is nearly sorted or containing identical values.

General Matesort Algorithm: The basic Matesort algorithm processes the data one bit at a time, the general radix Matesort processes the data one digit (a group of bits) at a time. A radix value of r implies that the digit size = log r, e.g. radix 16 uses a digit size of 4 bits. Thus, compared to binary Matesort, we observe the following. First, the BitPartition method is replaced by the DigitPartition method that is passed digitloc (digit location) and digitval (digit value) parameters. Second, for radix r and a given digitloc, the data must be split into r groups, one for each radix value. This process can be done sequentially, one digit value at a time (sequential partitioning), or through divide-and conquer.

Lemma 1: for random uniformly distributed data, the expected number of element accesses [swaps] performed by the general radix Matesort using sequential partitioning for the first digit location is

$$(r-1)\, n - (n/r)\, (r-2)(r-1)/2.$$

Lemma 2: for random uniformly distributed data, the expected number of element accesses [swaps] performed by the general radix Matesort using divideand- conquer partitioning for the first digit location is

$$(r-1)n\, [n/2 \log r].$$

Lemma 3: for random uniformly distributed data, the expected number of element accesses [swaps] performed by binary Matesort for the first bit location is n[n/2].

Conclusion: The author has presented and analyzed a number of in-place MSD radix sort algorithms, collectively referred to as Matesort algorithms. These algorithms are evolved from the classical radix exchange sort. Experiments have shown that binary Matesort is of comparable speed to Quicksort for random uniformly distributed integer data. Unlike Quicksort, which becomes slower as data redundancy increases and may degenerate into O(n2) algorithm, binary Matesort algorithm is unaffected and remains bounded by O(kn), where k is the element size in bits. We have discussed three partitioning methods for use by the general radix Matesort algorithm: sequential, divide-and-conquer and permutation-loop. For English text, experiments have shown that the general radix Matesort using divide and-conquer partitioning is the fastest. Moreover, the divide-and-conquer method can be optimized further to exploit data redundancy. Finally, the experimental results reported here show that Matesort (and also Quicksort) are much faster (twice as fast in many instances) than the built-in Microsoft .Net Array Sort method, which suggests that Microsoft ought to examine their implementation.

This survey is done on A new efficient radix sort paper.

Agenda: Introduced Forward Radix Sort which combines the advantages of traditional left-to-right and right-to-left radix sort in a simple manner. They argue that this algorithm will work very well in practice. Adding a preprocessing step, they obtain an algorithm with attractive theoretical properties.

Background: A common idea in the design of efficient algorithms is to take the distribution of the input into account. Well known examples are interpolation search [9, 21, 22], trie structures [8, 10, 13], and bucketing algorithms [6, 7]. When describing the complexity of such algorithms, it is common to rely on the assumption that the input elements are independently and randomly drawn from a specific distribution, for instance the uniform or normal distribution. A more general approach is to assume no particular distribution, but to express the complexity of the algorithm in terms of certain properties of the input.

The extended algorithm has favorable properties from a theoretical point of view. In particular, it is always possible to sort n strings in time proportional to the time to read the distinguishing prefixes plus the time to sort n integers of length w. In effect, the string sorting problem has been reduced to an integer sorting problem. Reading the distinguishing prefixes cannot be avoided, since these prefixes must be read to verify that the strings are sorted. However, there is a trade-off between the preprocessing step and the integer sorting step and it is often worthwhile to spend more time on the preprocessing to get a simpler integer sorting problem.

MSD and LSD algorithms differ in two major aspects. First, in the forward algorithm, we only need to scan the distinguishing prefixes, while the entire strings must be scanned in the backward algorithm. Second, the recursive application of the forward algorithm is made on the groups separately, while the strings are kept together in the backward algorithm. The first fact gives an advantage to the forward algorithm while the second fact gives an advantage to the backward algorithm.

Extended Algorithm: In the basic algorithm, we must visit all buckets, even the empty ones, in each pass. This may be avoided by a preprocessing step. During the preprocessing we create a list P of pairs. A pair (i,c) indicates that character c will split a group in pass number i. Using this idea we get an extended algorithm that consists of three steps.

1. Create a list P.
2. Sort list P.
3. Run the basic algorithm using P to avoid looking at empty buckets.

3. Implementation of MSD Radix sort for sorting Hindi words

For sorting Hindi words, we have to first convert words into their Unicode equivalent. In Unicode ASCII table Hindi words comes under Devnagri vowels and consonant. Devnagri ASCII code starts from 2309 and extend up to 2416.

Theorem:

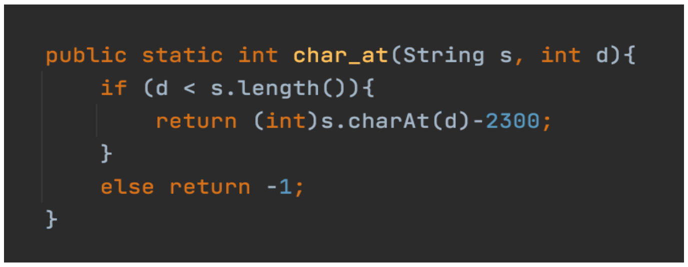
MSD radix sort algorithm takes O(nd) worst case running time where n is number of elements in the array arr[0… n] and d is the average length of strings.

Proof:

For this experiment we are taking 1 million Hindi words of variable size. d is the length of the strings which can extend from 1.. 5 character. MSD radix sort start sorting elements from left to right by putting together elements in the bucket who has the same key. And then recursively doing the same thing until all the strings are sorted.

3.1 Partitioning:

We need loop over 1.. n where n is the length of the array. For each d=0 i.e the first character in all strings to create bucket. This will act as as key for storing elements in count[] array. Count[] is an extra array with the length of n+1 needed to store keys. The time complexity for this action is O(n). Create an empty array count with the length of R. R is set to 400 considering Unicode value for Hindi words lies between 2300 and 2416.To get the value in the range 0 to 400 we are subtracting 2309 from character's index value.



```
public static int char_at(String s, int d){
    if (d < s.length()){
        return (int)s.charAt(d)-2300;
    }
    else return -1;
}
```

Fig.1 To get index of Hindi string character

3.2 Counting:

Next, we need to determine how many characters with the same key present in the array. For e.g In the count [] array first key is a and second is b. there are 2 strings with a as a key and 3 elements with b as a key. Now we have to count[r+1] = count[r+1] +

count[r]. this will tell us how many elements there before b which are shorter than b. the time complexity of this is O(R) where R is the Radix value. For Hindi words R is set to 2416 as Ascii values extend till 2416.

3.3 Sorting:

Further, we need to place the data to their correct position in the array calculated by their ascii key and stored in count[] array. For this purpose we need an extra empty array aux[]. This step will be needing time complexity of O(n).

3.4 Final step

This is the final step where we have to copy array from aux to original array arr[] for further recursion. At this step we need to make sure invariance in MSD is maintained. Invariance in MSD says After the ith iteration of the loop, the elements are sorted by their last i digits.The time complexity of this step would be O(n).

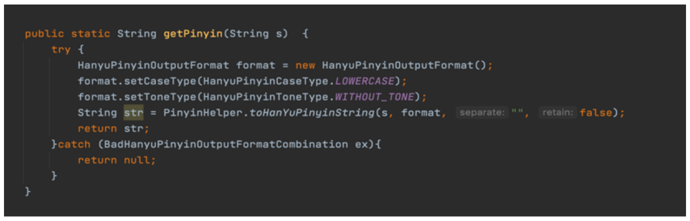
3.5 Recursion

Till now our array is sorted with the first character from the left. Repeat the steps from 2.1 to get the final sorted array. The time complexity of this steps would be depending upon the length of the largest string in the array. Lets consider d is the length of the largest string in the array, then final complexity of the sort is O(dn).

4. Techniques used for MSD Radix algorithm for sorting Chinese words

For sorting Chinese words we have taken Pinyin into consideration. Pinyin is a used to Romanize the Chinese characters for learning to speak the language faster. For this purpose, we have used Pinyin4j java library to convert Chinese character into Pinyin systems.

A common function getPinyin() is implemented which converts string into equivalent Unicode characters.



```
public static String getPinyin(String s)  {
    try {
        HanyuPinyinOutputFormat format = new HanyuPinyinOutputFormat();
        format.setCaseType(HanyuPinyinCaseType.LOWERCASE);
        format.setToneType(HanyuPinyinToneType.WITHOUT_TONE);
        String str = PinyinHelper.toHanYuPinyinString(s, format, separate: "", retain: false);
        return str;
    }catch (BadHanyuPinyinOutputFormatCombination ex){
        return null;
    }
}
```

Fig.2 Function to convert pinyin to Unicode string character

We have implemented a class named Pair<K,V> for storing key and value where key is pinyin Chinese word and value is converted Unicode equivalent using function getPinyin() as stated above.

Theorem:

The conventional order for Chinese is according to the English order of the Pinyin. Sort Chinese word with O(nd) time complexity while considering Pinyin order of Chinese characters.

Proof:

For this experiment we are taking 1 million Hindi words of variable size. d is the length of the strings which can extend from 1.. 5 character. MSD radix sort start sorting elements from left to right by putting together elements in the bucket who has the same key. And then recursively doing the same thing until all the strings are sorted.

4.1 Partitioning:

We need loop over 1.. n where n is the length of the array. For each d=0 i.e the first character in all strings to create bucket. This will act as key for storing elements in count[] array. count[] is an extra array with the length of n+1 needed to store keys. The time complexity for this action is O(n).

4.2 Counting:

Next, we need to determine how many characters with the same key present in the array. For e.g. In the count [] array first key is a and second is b. there are 2 strings with a as a key and 3 elements with b as a key. Now we have to

$$count[r+1]=count[r+1]+count[r]$$

This will tell us how many elements there before b which are shorter than b. the time complexity of this is O(R) where R is the Radix value. For Hindi words R is set to 2416 as Ascii values extend till 2416.

2.3 Sorting:

Further, we need to place the data to their correct position in the array calculated by their ascii key and stored in count[] array. For this purpose we need an extra empty array aux[]. This step will be needing time complexity of O(n).

2.4 Final step

This is the final step where we have to copy array from aux to original array arr[] for further recursion. At this step we need to make sure invariance in MSD is maintained. Invariance in MSD says After the ith iteration of the loop, the elements are sorted by their last i digits.The time complexity of this step would be O(n).

2.5 Recursion

Till now our array is sorted with the first character from the left. Repeat the steps from 2.1 to get the final sorted array. The time complexity of this steps would be depending upon the length of the largest string in the array. Lets consider d is the length of the largest string in the array, then final complexity of the sort is O(dn).

5. Implementation Results

To demonstrate the implementation, we have used a graph library XCharts to show the efficiency of algorithms. Different algorithms like Radix Sort, LSD Radix, Tim Sort, Quick Dual Pivot Sort and Husky sort is taken in consideration to observer the efficiency based on the size of the input provided. Currently the size are 250K, 500K, 1M, 2M, 4M.

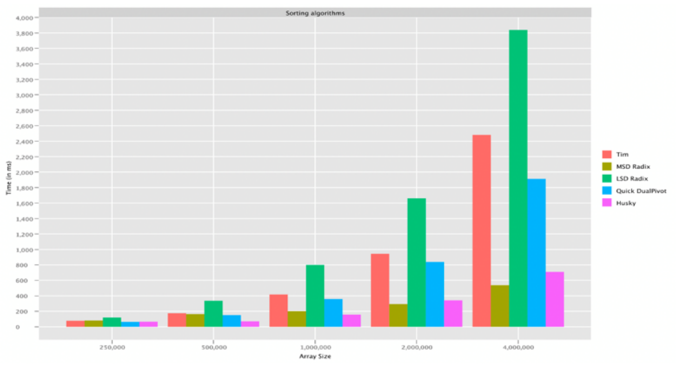


Fig.3 Bar graph showing different sorting techniques for Hindi words

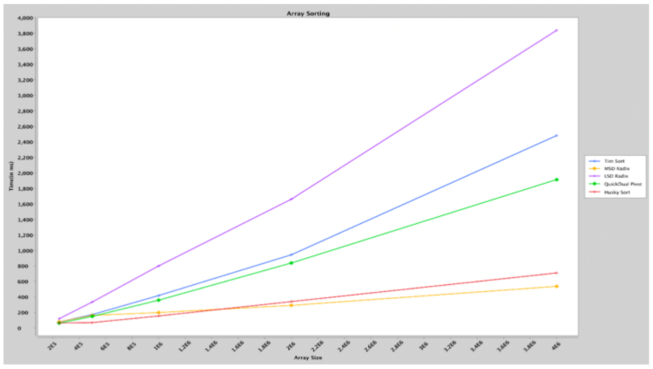


Fig.4 Line Chart showing different sorting techniques for Hindi words

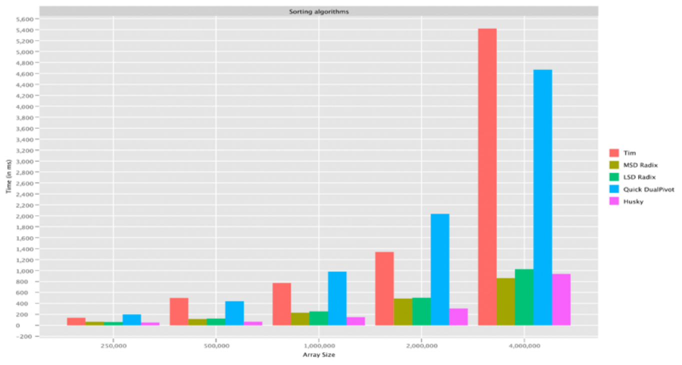


Fig.5 Bar graph showing different sorting techniques for Chinese words

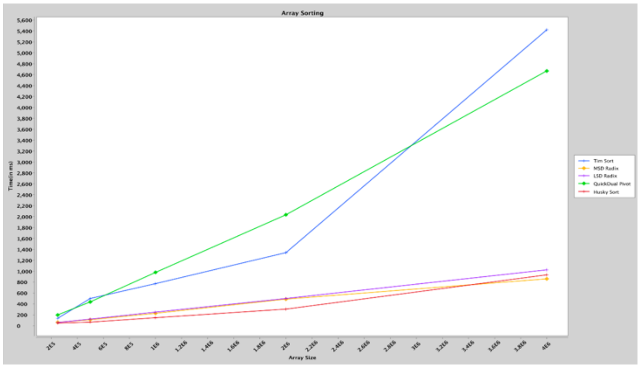


Fig.6 Line Chart showing different sorting techniques for Chinese words

6. Conclusion

For Hindi input

As the input size increase the time measurement for LSD radix has significantly increases, followed by Tim sort. The most efficient algorithm as input size increases are MSD and Husky Sort. LSD performance best when the length of the string in the array are equal. If the string length varies, LSD sort performance significantly decreases.

For Chinese input

Tim and quick sort complexity increase with the input size. Husky Sort is the most efficient followed by MSD Radix and LSD Radix.

7. Future Aspects

MSD radix algorithm is much too slow for small array size. It can be improved by switching to Insertion sort if the array size is comparatively small. Same implementation can be found in Husky Sort, for which performance is significantly efficient regardless of the array size.

8. References

[1] https://www.coursera.org/lecture/algorithms-part2/msd-radix-sort-gFxwG

[2] https://en.wikipedia.org/wiki/Pinyin

[3] https://unicode.org/charts/PDF/U0900.pdf

[4]https://sites.psu.edu/symbolcodes/languages/southasia/devanagari/devanagarichart

[5] https://www.javatpoint.com/tim-sort

[6] https://github.com/rchillyard/The-repository-formerly-known-as

[7] https://github.com/rchillyard/INFO6205