INFO6205 Final Project Report
Literature Survey

By : Akshaysingh Bayes (002956209) | Nimish Sharma (001598648) | Tanay Saxena (001586302)

Introduction:

In computer science, sorting algorithms are fundamental problems that have wide applications. As soon as run-time and space complexity were formalized, sorting algorithms were the logical candidate for analysis and improvement. There are two kinds of sorting algorithms: comparison-based and distribution-based. A comparison-based algorithm, like Heapsort or Quicksort, sorts by comparing two elements at a time. Alternatively, a distribution-based algorithm, such as radix sort, distributes the elements according to their values.

Radix sort algorithms can be divided into two categories: 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. As a result, such an algorithm uses O(n) space for the combined buckets and is capable of sorting n non-negative integers between [0,m] in O(n log m) order of running time.

MSD Radix Sort & Quicksort Review:

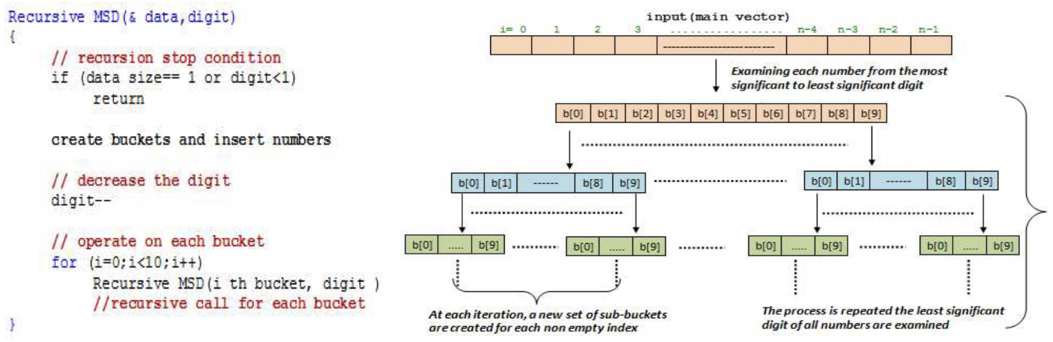
MSD Radix Sort (Recursive approach):

For the sake of simplicity and keeping in mind the primary goal that is to compare the different algorithms and optimization schemes, we are going to use Linked

Lists for our all implementations even though array based implementations have the advantage that permutations can be done in place with the only downside being the stability.

Since swapping the strings themselves could be an expensive operation, we would only be moving the pointers to them, which would in turn make each string operation take O(1) constant time. This is also a common practice to put each character of the string in consecutive memory locations and represent each of the strings by a pointer to the first character of the string.

The length of the string can be stored explicitly or the end of the string can be marked by a specially designated end-of-string character. A common well-known way of speeding up the algorithm is to use a comparison-based sorting mechanism on small subproblems (often Insertion Sort). We are using this technique in all the algorithms discussed in this review.



```
Recursive MSD(& data,digit)
{
    // recursion stop condition
    if (data size== 1 or digit<1)
        return

    create buckets and insert numbers

    // decrease the digit
    digit--

    // operate on each bucket
    for (i=0;i<10;i++)
        Recursive MSD(i th bucket, digit )
        //recursive call for each bucket
}
```

input (main vector)
i= 0
1
2
3
n-4
n-3
n-2
n-1
Examining each number from the most significant to least significant digit
b[0]
b[1]
b[2]
b[3]
b[4]
b[5]
b[6]
b[7]
b[8]
b[9]
At each iteration, a new set of sub-buckets are created for each non empty index
The process is repeated the least significant digit of all numbers are examined

Figure 1. Recursive MSD radix sorting process and pseudo code

We use an explicit stack to track the flow of computation, in order to avoid using a new bucket table for each invocation of bucket sort. First, we operate on the last bucket and push the sublists to be sorted onto the stack. The problem with this implementation is that the stack becomes sizable. However, it is quite easy to construct an example where the stack will contain only one entry for the key. The suggestion to overcome this problem is to sort both the lists on top of the stack, and the list to be pushed onto the stack so that there is no need to allocate another stack. We can simply append one list to the end of the other. [4] This technique is applicable if the algorithm switches to a simple comparison-based algorithm to sort short subsequences. If we choose to switch when at most k elements remain in a group, each stack entry will contain at least k elements and hence the total size of the stack will be at most n/k. In practice, however, the stack will typically be much smaller. This optimization is likely to improve the running time, and on top of that gives a considerable reduction in the stack side.

The fundamental problem that we faced while choosing the size of the alphabet is that the large alphabet reduces the total number of passes but increases the total number of buckets that are needed to be inspected. One can also use heuristic methods so that larger characters that can not be encoded into ASCII can also be incorporated. A way to do that could be to record the minimum and maximum characters encountered during the bucketing phase, and then only look at these buckets that fall in between the range of extreme values.

To take advantage of the clustering (which is often present in many kinds of data) we use a common character as a single unit when moving them into the bucket. Since we are using a linked-list representation, we are able to move the sublist of identical elements in constant time. Which simply improves the performance in real life scenarios.

MSD Radix Sort (Non-Recursive approach):

The way that MSD radix sort starts its sorting activity by inspecting numbers from the most significant digit, allows for the assignment of each of the numbers in their last buckets after the 1st iteration. Hence, each bucket can be sorted individually in this range. The first 1st & 2nd buckets are two data structures implemented as a two-dimensional vector. We also use a middle bucket to arrange individual arrays belonging to the first bucket (and/or second bucket) for the subsequent digits as described below. Before the sorting process begins, the maximum number of inputs is found and its digit count is calculated.

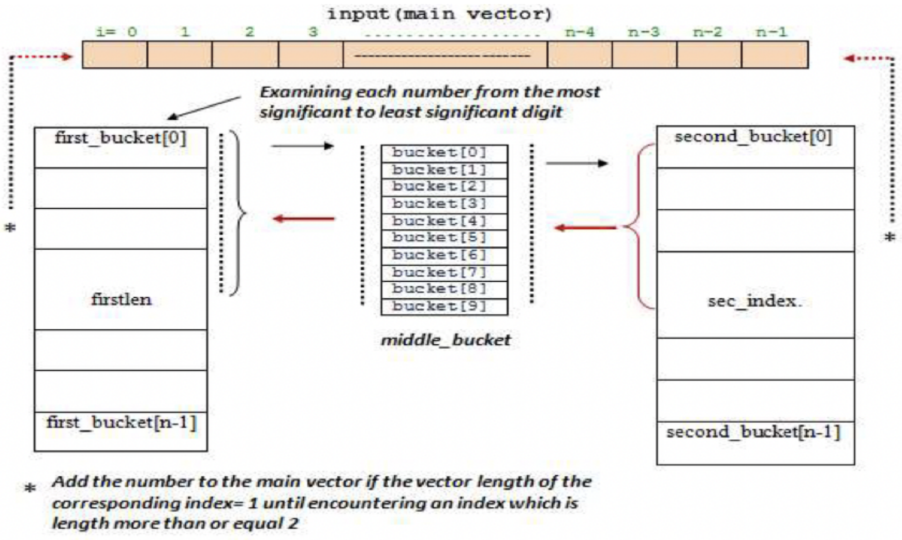


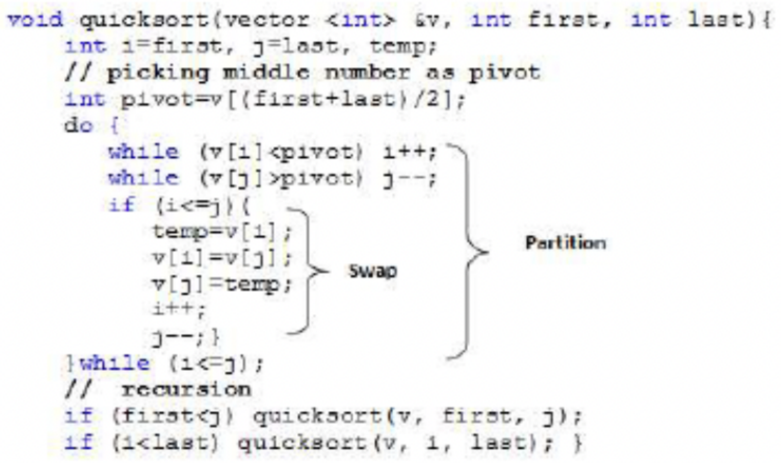
Figure 2: Non-Recursive MSD radix sort algorithm sorting process diagram

The Process starts by evaluating the MSD of each number in the primary array, it then adds each number with the corresponding digit value “i” ([0, 9]) in its bucket of the first bucket. After the first iteration, all numbers are copied in the first array. Each index in this structure is considered as an individual array which is sorted for the next most significant digit using the middle bucket. The content of the middle bucket is added to the second bucket, and the process is repeated until all indices of the first bucket are processed and the first bucket is empty. And so on until all the buckets are evaluated.

During this, numbers can be in either of the buckets. If, at each iteration, the index length is 1 (the middle bucket is not needed) the index can be added to the main array till we reach an index having a length greater than one. This reduces the index size in future iterations.

Multi-pivot Quicksort:

Sorting using Quicksort is a popular divide-and-conquer algorithm. lt is quite easy to implement, works well with a variety of input data types, and is significantly faster than any other sorting method. The quicksort algorithm’s desirable features are that it is in-place (uses only a small auxiliary stack) and that it requires time proportional to N log N on the average to sort an array of length N. However, its primary drawback is its fragility, as the implementation must be carefully controlled to avoid performance problems.



```
void quicksort(vector <int> &v, int first, int last){
    int i=first, j=last, temp;
    // picking middle number as pivot
    int pivot=v[(first+last)/2];
    do {
        while (v[i]<pivot) i++;
        while (v[j]>pivot) j--;
        if (i<=j){
            temp=v[i];
            v[i]=v[j];
            v[j]=temp;
            i++;
            j--;}
    }while (i<=j);
    //  recursion
    if (first<j) quicksort(v, first, j);
    if (i<last) quicksort(v, i, last); }
```

Figure 3: Quick Sort Algorithm implementation

Partitioning is the crux of the Quicksort algorithm and is implemented as an individual function (Cormen et al., 2009). Figure 3 depicts our implementation of Quicksort with the partitioning section included in the main procedure. While the selection of the pivot element is important in the performance tuning of Quicksort, we use the middle element in all of our tests. The performance can be improved considerably if the pivot is selected randomly or by using multiple pivots for sorting.

While using Quicksort with strings, it exhibits low performance with large arrays like those with more than one million elements. This is because Quicksort is a comparison-based algorithm that compares two elements at a time. Due to this, the algorithm compares each element one by one, thus lowering its performance.

Sequential MSD Radix Sort Versions Analysis

Ahmet Arif Aydin, and Gita Alaghband[4] conducted an experiment in their paper to benchmark the performance of the recursive MSD radix sort, non-recursive MSD radix sort, and Quicksort which is depicted in Figure 4. In this case, the input consists of 2 million randomly generated numbers having varying digit lengths.

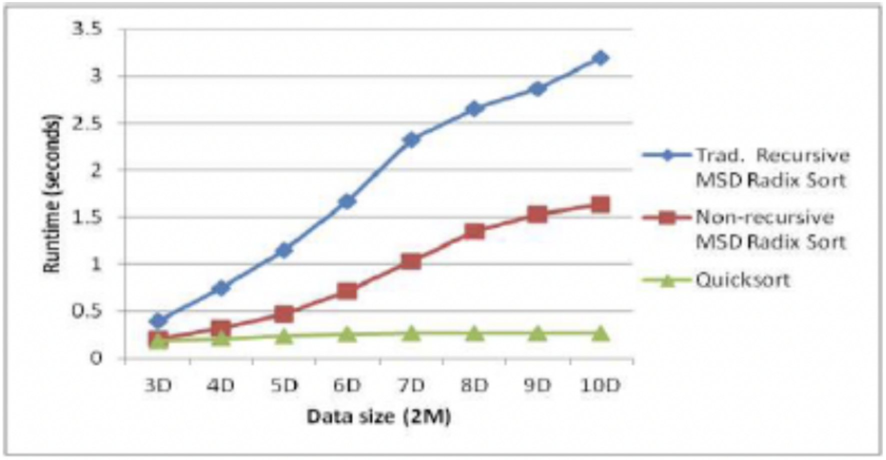


Figure 4: Performance comparison of sequential algorithms

Figure 4 shows that as the length of numbers increases from 3-10 digits, the runtime of MSD radix sort versions increases because their sorting process is related to the digit count of the input numbers. Moreover, we observe that Quicksort's performance is not significantly changed for different digit-count values making it an excellent choice for sorting the individual vectors produced after applying MSD for several steps In Figure 4, the non-recursive MSD radix sort version runs 1.74 to 2.56 times faster than the traditional recursive MSD radix sort.

Conclusion:

An efficient sequential non-recursive MD radix sort version is implemented as an alternative version to the recursive one. Results presented show the non-recursive MD radix sort performs remarkably better than the traditional recursive MSD radix sort. Moreover, the non-recursive MSD radix sort and the efficient Quicksort methods are used to implement two versions of sequential MSD radix sort versions. Performance benchmarks of sequential MSD radix sort algorithms also reveal a significant performance improvement over multi pivot Quicksort while sorting large-element input arrays.

References:

[1] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. Comp. Systems, 6(1):5{27, 1993.

[2] Amer Badarneh, and Fouad El-Aker, Efficient Adaptive In‑Place Radix Sorting, INFORMATICA, 2004, Vol. 15, No. 3, 295–302 295 2004

[3] Nasir Al-Darwish, Formulation, and analysis of in-place MSD radix sort algorithms, Journal of Information Science 31(6):467-481

[4] Ahmet Arif Aydin, and Gita Alaghband, Sequential and Parallel Hybrid Approach for Non-Recursive most significant digit radix sort, IADIS International Conference Applied Computing 2013.

[5] Arne Andersson, and Stefan Nilsson, Implementing Radix Sort , http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.4536&rep=rep1&type=pdf