**ANALYSIS OF RADIX SORT ALGORITHMS**

INFO6205 Program Structures and Algorithms  
Course Professor: Robin Hillyard

Abhishek Ravindra Satbhai 001587689  
  
Apoorva Kethamaranahalli 001090928  
  
Sneha Ravichandran 001096251

**Exploring Radix Sort**

**Abstract:** Radix sort is an efficient non-comparative integer sorting algorithm which can be generalized to floating point numbers and strings of characters. We try to understand how efficient MSD Radix sort is when compared to other stable comparison sorts and explore new optimization and implementation techniques for String sorting. We will be discussing about the Forward radix sort that has good worst-case behavior. According to experimental results we found out that Radix sorting is comparatively faster in comparison to comparison-based methods. This is true even for small inputs.

**Introduction:**

Radix sort is a simple and very efficient sorting method especially for strings.

The MSD radixsort algorithm works as follows.

At first, we split the strings into groups according to their first character or the Most significant digit and arrange the groups or buckets in ascending order.

Second, we apply the algorithm recursively on each bucket separately, disregarding the first character of each string, the groups containing only one string will not be processed further..

The advantage of this algorithm is that it inspects only the distinguish prefixes and avoids redundancy, that is the minimum number of characters that has to be inspected to assure that the strings are, in fact, sorted

The problem with MSD radixsort is that it might check many number of empty buckets. The 2 techniques to overcome this problem is

1) To switch to a comparison-based sorting algorithm, such as Insertion sort, Husky sort and Tim sort for small subproblems and to use a heuristic, such as keeping track of the minimum and maximum character that is encountered in one round, to reduce the number of buckets that needs to be checked.

2) Discussed in the paper, it takes a different approach and studies two algorithms: Adaptive radixsort and Forward radixsort. Adaptive radixsort is a modification of simple MSD radixsort where the size of the alphabet is dynamically chosen. This algorithm gives us good results. Forward radixsort [1] combines the advantages of LSD and MSD radixsort. It scans the input starting with the most significant digit but performs bucketing only once per character position. It has a good worst-case complexity and it runs as fast as Adaptive radixsort.

Adaptive radix sort:

Forward Radix Sort:

To avoid bad worst-case performance due to fragmentation of the data into many small sublists. It combines the advantages of MSD and LSD. The main plus point of LSD radixsort is that it checks a complete string horizonatally at a time; the main weakness is that it inspects all characters of the input. MSD radixsort only inspects the distinguishing prefixes of the strings, but it does not make efficient use of the buckets. The Forward radixsort explained in the paper starts with the most significant digit, performs bucketing only once for each entry, and inspects only the significant characters.

The invariance is maintained which is that after the jth pass all the strings are sorted according to the first j characters. The sorting is performed by splitting and breaking the strings into groups. Initially all string are in the same group one.  This group is split into smaller groups after each pass. Each group is also associated with a rank which is Each group is associated with a number that indicates the rank in the sorted set of the smallest string in the group. We also distinguish between finished and unfinished groups. A group will be finished in the ith pass if it contains only one string or if all the strings in the group are equal and not longer than i.

A pass in the algorithm is performed in the following steps:

1)Traverse groups that are not finished in ascending order and insert each string x, tagged by its current group number, into bucket number x(i), where x(i) is the ith character of x.

2)Go through the buckets in ascending order and put the strings back into their respective groups in the order they are put into the buckets. If all checked characters are equal, no bucketing is performed.

3) We traverse the groups separately. If the kth string in group g differs from its predecessor in the ith character then we split the group at this string. The new group is numbered g + k -1.

We use a linkedlist to keep track of groups, each entry contains list of elements with same prefixes. A pointer is used to indicate the start of the next incomplete group. This helps us breakdown the group in constant time. The groups or buckets are implemented as an array of linked lists, one for each character in the alphabet. Each bucket will contain lists of elements with the common tag stored in the head of the list, a new entry bucket is created on if the tags are different from the list that is to be put into the bucket.

The paper has implemented the algorithm with two different alphabets: 8-bit and 16-bit characters. For the 16-bit and 8-bit alphabet we avoid inspecting too many empty buckets. Finally, the algorithm checks, in each pass, if all inspected characters are equal. If this is the case, no bucketing is performed.

The picture below shows the buckets after the insertion of all remaining strings with prefix \SO."

Diagram

Description automatically generated

The one below is the group list after 3 passes

Diagram

Description automatically generated

The algorithm will always insert an element into the first list of the bucket/group and hence the insertion time for this algorithm is constant. The paper also discusses about switching to simple comparison based sorting such as insertion sort which eventually helps with the space complexity and reduces the running time.

**Complexity Discussion:**

The forward radixsort runs in Big O( O(S + n +m. Smax )) time, where S is the total characters of the distinguishing prefixes, Smax is the length of the longest distinguishing prefix, and m is the size of the alphabet. The first two terms come from the fact that the algorithm inspects each distinguishing character once and each string at least once. The algorithms checks m buckets in each pass and runs in Smax passes, this is how we arrive at the last term in the expression.

The worst-case running time is also bounded by O(S + n + m2 ). To see this, recall that the algorithm does not perform any bucketing in a pass where all the inspected characters are equal. We study two cases. First, when the number of strings remaining in the incomplete groups is larger than m the cost of bucketing is no larger than the cost of reading the characters. Second, when less than m strings remain there can be at most m passes in which any splitting of groups occurs and hence the total number of buckets traversed is O(m2 ) in this case.

**Experimental Results:**

The paper has implemented the algorithms with linked lists and they have refrained from micro-optimizations such as loop enrolling and manual register allocation.

Several researchers have independently found that a carefully coded radix sorting algorithm performs better than the comparison-based algorithms by a good margin [6, 14] and we have arrived at the same conclusion. According to the paper Forward radixsort a larger alphabet size yields an improvement already for about 5000 elements.

The below graph describes the time required for the Forward Radix sort in comparison to MSD Radix sort ,Quicksort and other comparison sort.

A picture containing diagram

Description automatically generated

Forward radixsort is the most robust algorithm. Even in this \worst-case" setting a straightforward radix sorting algorithm runs only slightly slower than Quicksort.

The important optimization for the sorting algorithms discussed in this paper is to switch to a simple comparison-based algorithm such as insertion sort when only a small number of elements remain to be sorted.

The precise value of the turn over point is not crucial. Values in the range of 10 to 30 gave satisfactory results. For Forward radixsort this optimization reduces the running time by about 40%.

Forward radixsort had a problem which was the large space overhead. However, the paper describes a simple and efficient solution to this problem: we use only one tag per group, instead of one tag per element. This idea alone reduces the total number of tags to less than 10% of the number of elements. Also, the running time decreases by roughly 10%. If we are more restrictive with the splitting and do not split consecutive groups that are already sorted. This simple optimization gives a drastic reduction in space complexity. The total number of buckets that are allocated is typically only 6% of the total number of elements. This optimization also reduces the running time by another 4%. With these two optimizations the space complexity of Forward radixsort is very good.

**References for the Literature Survey paper:**

https://stanford.edu/~rezab/classes/cme323/S16/projects\_reports/he.pdf

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.4536&rep=rep1&type=pdf

[1] A. Andersson and S. Nilsson. A new ecient radix sort. In Proceedings of the 35th Annual IEEE

Symposium on Foundations of Computer Science, pages 714{721, 1994.

[2] T. C. Bell, J. G. Cleary, and I. H. Witten. Text Compression. Prentice Hall, 1990.

[3] J. L. Bentley and M. D. McIlroy. Engineering a sort function. Software|Practice and Experience,

23(11):1249{1265, 1993.

[4] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In Proceedings of

the Eight Annual ACM-SIAM Symposium on Discrete Algorithms, pages 360{369, 1997.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. McGraw-Hill, 1990.

[6] I. J. Davis. A fast radix sort. The Computer Journal, 35(6):636{642, 1992.

[7] L. Devroye. Lecture Notes on Bucket Algorithms. Birkhuser, 1985.

[8] W. Dobosiewicz. Sorting by distributive partitioning. Information Processing Letters, 7(1):1{6, 1978.

[9] G. Ehrlich. Searching and sorting real numbers. Journal of Algorithms, 2(1):1{12, 1981.

[10] http://www.cs.hut.fi/~sni/radixexpe/

[11] G. H. Gonnet and R. A. Baeza-Yates. Handbook of Algorithms and Data Structures. Addison-Wesley,second edition, 1991.

[12] http://gutenberg.etext.org/

[13] J. H. Kingston. Algorithms and Data Structures: Design, Correctness, Analysis. Addison-Wesley, 1990.

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

[15] K. Mehlhorn. Sorting and Searching, vol 1 of Data Structures and Algorithms. Springer-Verlag, 1984.

[16] M. Tamminen. Analysis of N-trees. Information Processing Letters, 16(3):131{137, 1983.

[17] The Unicode Consortium. The Unicode Standard: Worldwide Character Encoding. Version 1.0, volume

1 and 2. Addison-Wesley, 1991.[18] https://www.cs.princeton.edu/courses/archive/spr15/cos226/lectures/51StringSorts.pdf