ISBN 978-981-11-0008-6
Proceedings of 2016 6th International Workshop on Computer Science and Engineering (WCSE 2016)
Tokyo, Japan, 17-19 June, 2016, pp. 613-617
doi: 10.18178/wcse.2016.06.106

SONIK: Efficient In-Situ All Item Rank Generation Using Bit Operations

Sourav Dutta [+]

Max Planck Institute for Informatics, Germany

Abstract. Sorting, a classical combinatorial process, forms the bedrock of numerous algorithms with varied applications. A related problem involves efficiently finding the corresponding ranks of all the elements – catering to rank queries, data partitioning and allocation, etc. Although, the element ranks can be subsequently obtained by initially sorting the elements, such procedures involve O(n log n) computations and might not be suitable with large input sizes for hard real-time systems or for applications with data re-ordering constraints. This paper proposes SONIK, a non-comparison linear time and space algorithm using bit operations inspired by radix sort for computing the ranks of all input integer elements, providing implicit sorting. The element ranks are generated *in-situ* without re-ordering or using other sorting mechanism.

Keywords: all item rank computation, integer sort, non-comparison sort, bit operations, linear time.

1. Introduction and Related Work

Preliminaries: Given a collection of n integer elements, $\sigma = \{e_1, e_2, \cdots, e_n\}$, sorting refers to the procedure of arranging the elements in accordance with a definitive ordering, e.g., numeric or lexicographic. Mathematically, sorting produces a permutation of the input items such that the elements in the final list are ordered as $e_i \leq e_j$, for all $i \leq j$. The literature provides an enormous wealth of applications involving sorting, as a major operational step, such as comparison and searching, statistical analysis, and priority-based scheduling to name a few. Advanced algorithms such as knapsack problem [18, 7], minimum spanning tree [4, 9, 17], and network analysis also employ sorting as an intermediate stage. Several sorting algorithms such as bubblesort [16, 5], quicksort [14], mergesort [16], heapsort [21], etc., based on different strategies such as divide-and-conquer, exchange, partitioning, and greedy approach [5] have been proposed.

Sorting algorithms can be classified into two categories – comparison sort (such as mergesort, heapsort, etc.,) involving element-wise comparison bounded by *O(n log n)* [5], or counting based sort using the count of items bounded by *O(n)* assuming certain characteristics of the input data such as uniform distribution (bucket sort [5, 6]) or small range of input element (counting sort [8]), where *n* is the number of elements. Sorting algorithms are also characterized by behavioral properties, such as *in-place*, *stable* [5], and *adaptive* [19]. Several parallelized procedures such as bitonic sort [2], odd-even sort [11], etc. have also been studied.

Motivation: A closely related problem involves the computation of ranks of all the elements in an input list, where the rank of an item is defined as its position in the final sorted list. For example, given an input collection $\sigma = \{7, 2, 1, 5, 4\}$ the corresponding rank list for the elements is $\rho = \{5, 2, 1, 4, 3\}$, based on the non-decreasing key sorted order $\{1, 2, 4, 5, 7\}$ of σ. In fact, several applications involving scheduling, order statistics, job priority, data partitioning, etc., [5, 16] use element ranks as the working metric. For example, consider the scenario of task scheduling from a job queue within a server pool with heterogeneous configurations, and the rank of a job to denote its priority. Unfortunately, limited work exists in this regard.

Application: Multi-criteria searching require the input objects (represented by a d-dimensional vector of

[+] Corresponding author. Tel.: +004917641549982.
E-mail address: sdutta@mpi-inf.mpg.de..

integers) to be ordered such that most (or all) of the d-dimensional values are nearly (or completely) sorted. As such, the objects need to be independently sorted based on their i^{th} dimension, and then suitably combine into a final sorted list of the objects with most of the dimensions nearly sorted, using expensive join. However, an efficient approach finding the ranks of all objects for the individual dimensions in a pre-defined or input appearance order would eliminate the need for the expensive merge procedure. This paper tackles the *all-item ranking problem* to compute a rank list of input elements in appearance order, i.e., in-situ.

Problem Statement: Formally, given an input list of n elements, $\sigma = \{e_1, e_2, \cdots, e_n\}$, the problem of all-item ranking entails the efficient computation of the rank list $\rho = \{r_1, r_2, \cdots, r_n\}$, where r_i denotes the rank of element e_i, $i \in [1, n]$, with rank r_i referring to the position of element e_i in the sorted order of σ.

State-of-the-art: Naïvely, we could initially sort the input list and thereafter obtain the ranks of individual elements by searching its position within the ordered list. Observe that, sorting of the input list can be performed in *O(n)* best case by use of non-comparison based sorting approaches. Moreover, computation of the rank list for the input elements (in the order of appearance in the input list) involves subsequent searching of each of the elements in the sorted order for their positions (i.e., ranks). For example, in our previous example, for obtaining the rank of $e_1 = 7$ requires the searching of e_1 in the sorted list {1, 2, 4, 5, 7}. This involves additional *O(log n)* computations for each element, using binary search procedure. Hence, the total complexity for obtaining the rank list is bounded by *O(n log n)*.

The quickselect algorithm [15] working on the divide-and-conquer principle provides the state-of-the-art O(n) approach to obtain the item with rank *r* from an unsorted list. However, finding the ranks of all elements degenerates its performance to quadratic complexity, i.e., $O(n^2)$, for finding the ranks of all items.

To alleviate such problems, the augmented order-statistics tree [5] was proposed to efficiently store and index the input elements in a tree-based structure by storing extra statistics at each node of the tree (e.g., number of descendants in the left subtree, etc.). Subsequent querying for the item having rank *r* can then be performed in O(h), where *h* represents the height of the tree structure. The use of height-balanced tree data structures like red-black tree [10] or AVL tree [1] provides an upper bound of O(log n) for *h* and and for finding the item with rank *r*. Thus, rank computation for all the elements (i.e., rank list) therein can be performed in O(n log n) time. However, such approaches suffer from involved tree rotational procedures for height balancing and also incur huge space requirements due to extra information and pointer storage. Hence, state-of-the-art computation of ranks of all input elements can be performed in O(n log n).

Contribution: In this paper, we propose the SONIK algorithm for the all-item ranking problem, i.e., finding the rank list for an input list of integer elements in their input appearance order, in *O(nk)* time, where *n* is the number of input elements and *k* denotes the number of bits required to represent the elements. Considering, *k* to be constant we obtain a linear time algorithm, and show that the space requirement is also linear in *n.* SONIK employs bit comparison operations (similar to the working of LSB radix-sort [13]) to provide an efficient approach for real-time scenarios, and also an *anytime algorithm* [12] obtaining the partial ranks of elements based on the *i* least significant bits after *i* iterations. We also show that no re-ordering of elements or dependence on other sorting approaches (as in radix-sort) is required by our approach. Further, the generated element ranks provide an inherent sorting of the input list in linear time.

2. SONIK Algorithm

We assume the input list of elements, σ, to consist of non-negative integers without the presence of any duplicates, and that the item ranks are to be computed based on the non-decreasing sorted order of σ.

As a running example, consider $\sigma = \{7, 2, 1, 5, 4\}$ to be the input list and $\rho = \{r_1, \cdots, r_5\}$ as the final element rank list to be computed, where r_i represents the rank of element $e_i \in \sigma$. Further, let *k* be the number of bits required to encode the elements in σ; hence k = 3 for our above example.

The rank list, ρ is at first initialized to 1 with processing proceeding from the least-significant bit (LSB) to the most-significant bit (MSB) of the bit encoding of input items, $\sigma_{bit} = \{111, 010, 001, 101, 100\}$. SONIK iterates over the *k* bits of the input elements, and at each iteration performs two major computational phases – (1) *Rank Updation*, and (2) *Rank Consolidation*.

At the onset of iteration *i*, the i^{th} least significant bits of all the elements in σ are extracted and two lists, L^i_0 and L^i_1, are created based on the associated bit values of 0 and 1 respectively, to store the current ranks.

Observation 1. Given only the least significant *i* bits of non-negative integers, elements with the i^{th} bit set (i.e., elements in list L^i_1) have a higher numerical value compared to the elements associated with list L^i_0 (having i^{th} bit value as 0), and hence should have larger ranks (based on non-decreasing sorted order of σ).

For our example, during the first iteration elements {7, 1, 5} ϵ σ have the rightmost bit set to 1 (from σ_{bit}) and hence their corresponding ranks are used to obtain list L^1_1 as {1, 1, 1} . Similarly, corresponding to the remaining elements, we obtain L^1_0 = {1, 1}. Based on the above observation, considering only the LSB, elements {7, 1, 5} (of list L^1_1) should be assigned larger ranks as compared to elements {2, 4} (in list L^1_0). The rank updation and rank consolidation phases are then performed separately for lists L^i_0 and L^i_1 as:

2.1. L_0 list processing

The ranks of the elements corresponding to the list L^i_0 (obtained above) are processed by SONIK as:

Rank Updation Phase: Using the above computed lists, L^i_0 and L^i_1, at each iteration the rank updation phase computes the following three operating variables:

- π^i_m :– denotes the minimum rank *m* among the elements having the i^{th} bit position set to 1, i.e., minimum value present in list L^i_1;
- $\pi^i_{\leq m}$:– depicts the maximum rank lesser than or equal to π^i_m among the elements corresponding to list L^i_0, i.e., elements with i^{th} bit set to 0. The value of $\pi^i_{\leq m}$ is considered to be 0 if no such rank is found;
- $\pi^i_{\geq m}$:– represents the minimum rank greater than or equal to π^i_m present in list L^i_0 , i.e., among the ranks of elements with i^{th} bit set to 0. Alternatively, this value is set to $\pi^i_{\leq m}$ if no such rank is found.

From Observation 1, based on the i^{th} bit, the elements associated to list L^i_0 having a rank greater than the minimum rank among the elements of L^i_1 (π^i_m), are processed during the rank updation stage. On the other hand, ranks of elements of L^i_0 with current ranks less than or equal to π^i_m remain unchanged during this step. The element(s) in L^i_0 with rank $\pi^i_{\geq m}$ are now re-assigned a rank of π^i_m, currently the numerically smallest element previously assigned a higher rank than π^i_m (in list L^i_1) during the iterations based on *(i−1)* LSB. However, no such comparative deductions can be made for elements in L^i_0 that have the same rank value of π^i_m (i.e., when $\pi^i_m = \pi^i_{\leq m}$). In these cases, the rank of such element(s) is kept unchanged during the current iteration and its final rank is computed by subsequent iteration steps.

Hence, for each element in list L^i_0 having a rank greater than π^i_m, SONIK decrements the current rank by the rank update parameter value, δ_0, defined as follows:

$$\delta_0 = \begin{cases} 0 & \text{if } \pi^i_m = \pi^i_{\leq m} \\ \pi^i_{\geq m} - \pi^i_m & \text{otherwise} \end{cases} \quad (1)$$

Thus, the updated element ranks are obtained by,

$$\rho_i = \begin{cases} \rho_i & \text{if } \rho_i \leq \pi^i_m \\ \rho_i - \delta_0 & \text{otherwise} \end{cases} \qquad [\forall i, e_i \in L^i_0] \quad (2)$$

The updated ranks of elements in list L^i_0 are then provided as input to the rank consolidation phase.

Rank Consolidation Phase: The consolidation stage processes the ranks to make them contiguous, to eliminate "holes". For example, an input rank list τ = {1, 2, 4, 6} is consolidated to τ' = {1, 2, 3, 4}.

Since the ranks of all the candidate items (in L^i_0) are reduced by a common factor, the original ordering among these elements are inherently preserved. Further, since the re-assigned ranks are greater than π^i_m, and element ranks less than π^i_m are not modified, the sorted property is also preserved for the updated rank list.

We now show that with the use of an extra storage space and two array position pointers, the rank consolidation phase can be performed in linear time in the input rank list size, with additional array E of size 2*n, and each element t_i ϵ τ is mapped to the array position E[t_i]. The array position pointer, p_1 provides the next free slot, while the other position pointer p_2 refers to the next rank to be consolidated. The rank value of E[p_2] is then accordingly set at position E[p_1] based on the value at E[p_1 −1].

For our example list, we create and initialize E[8] = {1, 2,−, 4,−, 6} with the pointers initially set as p_1 = p_2 = 1. We obtain p_1 = 3 and p_2 = 4. The array E is then updated as E[p_1] = E[p_1 − 1] + 1 and E[p_2] = −, to obtain {1, 2, 3,−,−, 6}. Finally, the first |τ| elements of E = {1, 2, 3, 4,−, −} are returned with a single pass.

2.2. L_1 List Processing

The ranks of the elements corresponding to list L^i_1 are now adjusted based on *max*, the maximum rank assigned to an element during rank consolidation on list L^i_0. Observe that, currently the minimum ranked element in L^i_1 (with rank of π^i_m) is now assigned a rank of *max+1*, and hence the current ranks of all elements in list L^i_1 are incremented by the update parameter, δ_1 = max+ 1− π^i_m. Hence, $\rho_i = \rho_i + \delta_1$ (for all *i*, $e_i \epsilon L^i_1$).

The obtained output ranks for list L^i_1 from the above rank update procedure are then provided to the rank consolidation step for generating contiguous rankings, similar to the procedure described in Section 2.1.

Observation 2. The maximum rank obtained by an element during rank updation is bounded by 2*n.

The maximum rank value, *max* returned during consolidation on L^i_0 and correspondingly the rank update parameter, δ_1 = max+1 − π^i_m = n − 2 + 1 − 1 = n − 2. Hence, rank updation on list L^i_1 might generate a maximum of n + δ_1 = 2 * n − 2 as the rank for and element in L^i_1.

Overall Computation: The element rank list, ρ_i, obtained from the above computations by SONIK over the two lists L^i_0 and L^i_1, depicts the intermediate ranking output at the end of the i^{th} iteration. The rank list, ρ_i is then fed as an input for processing during the next iteration step. This procedure is repeated until all the *k* bits of the input elements in σ have been processed, and the final rank list, ρ.

Working Example: Continuing our previous example from Section 2, for the input element list σ = {7, 2, 1, 5, 4}, during the first iteration, list L^1_1 contains the current ranks of elements {7, 1, 5} and we obtain the update parameter δ_0 = 1 for rank updation for list L^1_0 returning *max=1* after consolidation. Hence the update parameter δ_1=1is used for re-assigning the element ranks in list L^1_1. Finally, after consolidation of L^i_1, the obtained rank list, ρ_1 forms the input for the next iterations, wherein a similar procedure is followed for the remaining bits. Finally, during the last iteration, the rank updation of L^3_1 generates a rank list of {6, 4, 3}, for the elements {7, 5, 4}, based on the update parameter δ_1 = 2. The rank consolidation phase now provide a contiguous rank list, removing ranking "holes" in L^1_1 to generate a consolidated rank list as {5, 4, 3}.

Discussion: SONIK thus provides an efficient linear time and space approach (discussed in Section 2.3) for computing the ranks (based on the non-decreasing sorted order) of all the input elements. Observe that our approach does not involve any item re-ordering or dependence on other sorting techniques.

Interestingly, SONIK depicts an *anytime algorithm*, wherein the rank computations can be terminated at any iteration *i*, and the current rank list ρ_i provides the ranking based on only the *i* least significant bits.

Finally, the output rank list for the input elements can be utilized to generate a sorted order of the input items. This can be performed in a single pass over the input and the rank lists (i.e., *O(n)* time).

2.3. Performance Analysis

Assume an input element list σ to be composed of *n* integers, with each element represented by *k* bits. We observe that at iteration *i*, the i^{th} least significant bit of the input elements are extracted and the lists L^i_0 and L^i_1 are constructed, performed in O(n) time with a single pass over the input list, σ. Each element contributes an entry in either of the two lists and hence the combined size of the lists is $|L^i_0 \cup L^i_1| = |\sigma| = n$.

The two phases of SONIK – rank updation and rank consolidation – are invoked separately on the two lists, and hence each of the procedures operates on a total of *n* elements. The rank updation process appropriately updates the rank using the updation parameters, δ_0 and δ_1. Thus, the time taken is bounded by the total size of the lists, i.e., O(n). The rank consolidation procedure, for each of the lists, creates an additional storage E of size 2*n and consolidates the input element ranks by a single pass over E, with a time complexity of O(n). The computation of the operating variables can also be computed in linear time by two iterations over the lists L^i_0 and L^i_1. Hence, the total time taken at each iteration *i* is bounded by *O(n)*, considering assignments and simple mathematical computations to be constant time operations.

Since, the total number of iterations is *k* (number of bits for input elements), the total running time

complexity of SONIK thus becomes *O(nk)*. Considering *k* as a small constant (in the order of few tens) in most practical scenarios, the run-time of SONIK can be considered to be linear in the size of the input list.

Also, the space complexity of SONIK is linear in the number of input elements, as the space required by the lists and the extra storage during rank consolidation is bounded by O(n). Hence, our proposed algorithm provides a time and space efficient real-time approach for computing the ranks of input elements.

3. Conclusions

This paper presented SONIK, a linear time and space algorithm for computing the rankings (based on sorted order) for all elements of an input list. Our algorithm uses bit operations to compare the numerical values of elements for assigning appropriate ranks, based on two operational steps – rank updation and rank consolidation. SONIK provides a linear-time technique to efficiently generated element ranks in the order of appearance in the input list, and hence eliminates the need for item re-ordering or intermediate sorting.

4. References

[1] G. G. Adelson-Velsky and E. Landis. An algorithm for the organization of information. *USSR Academy of Sciences*, 146:263–266, 1962.

[2] K. E. Batcher. Sorting Networks and their Applications. *AFIPS Joint Computing Conference*, 32:307–314, 1968.

[3] O. Boruvka. O Jistém Problému Minimálním (About a Certain Minimal Problem). *Práce Mor. Přírodoved*. Spol. v Brne III, 3, 1926.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT & McGraw-Hill, 2001.

[5] E. Corwin and A. Logar. Sorting in linear time — variations on the bucket sort. *Journal of Computing Sciences in Colleges*, 20(1):197–202, 2004.

[6] T. Dantzig. *Number: The Language of Science: A Critical Survey Written for the Cultured Non-Mathematician*. Macmillan, 1930.

[7] J. Edmonds. *Counting Sort, How to Think about Algorithms*. Cambridge University Press, pages 72–75, 2008.

[8] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[9] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. *Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[10] N. Haberman. Parallel Neighbor Sort (or the Glory of the Induction Principle). *CMU Technical report*, 1972.

[11] J. A. Hendler. *Artificial Intelligence Planning Systems*. Elsevier, 1992.

[12] H. Herman. *Art of Compiling Statistics*, 1889. Patent US395781(A).

[13] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of ACM*, 4(7):321–322, 1961.

[14] C. A. R. Hoare. Algorithm 65: Find. *Communications of ACM*, 4(7):321–322, 1961.

[15] D. Knuth. *The Art of Computer Programming, volume Third: Sorting and Searching*. Addison-Wesley, 1998.

[16] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *American Mathematical Society*, 7(1):48–50, 1956.

[17] G. B. Mathews. On the partition of numbers. *London Mathematical Society*, 28:486–490, 1897.

[18] O. Petersson and A. Moffat. A framework for adaptive sorting. *Lecture Notes in CS*, 621:422–433, 1992.

[19] J. Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[20] J. W. Williams. Algorithm 232 - Heapsort. *Communications of ACM*, 7(6):347–348, 1964