Linear Time Sorting of Skewed Distributions*

Edleno Silva de Moura [†]
Depto. de Ciência da Computação
Univ. Federal de Minas Gerais, Brazil
www.dcc.ufmg.br/~edleno

Gonzalo Navarro [‡]
Depto. de Ciencias de la Computación
Univ. de Chile, Chile
www.dcc.uchile.cl/~gnavarro

Nivio Ziviani [§]
Depto. de Ciência da Computação
Univ. Federal de Minas Gerais - Brazil
www.dcc.ufmg.br/~nivio

Abstract

This work presents an efficient linear average time algorithm to sort lists of integers that follow skewed distributions. It also studies a particular case where the list follows the Zipf's distribution, and presents a example application where the algorithm is used to reduce the time to build word-based Huffman codes.

1 Introduction

Sorting a generic list of numbers is a well studied problem that can be efficiently solved by using generic algorithms, such as *quicksort* [Hoa62], and *shellsort* [She59]. However, a generic algorithm may not be the best choice when the list to be sorted has some initial order, for example, when many elements in the list are already sorted. An alternative in these cases is to use adaptive algorithms, which take advantage of the partial order of the list to accelerate the sorting process [ECW92]. A sorting algorithm is adaptive if it sorts sequences that are close to sorted faster than random sequences, doing it without knowing how far the list is from the sorted sequence [PM95]. Therefore, the adaptive algorithms do not take advantage of previously known characteristics about the distribution of the elements.

*This work has been partially supported by PRONEX grant 76.97.1016.00 and CYTED VII.13 AMYRI Project.

[†]This work has been partially supported by CAPES scholarship

[‡]This work has been partially supported by Fondecyt grant 99-0627

[§]This work has been partially supported by CNPq grant 520916/94-8

Another alternative to reduce the sorting times is to change the model used to determine the key order. Most of the classic sorting algorithms work under the "comparison based" model, i.e., they sort the list exclusively through pairwise comparison. However, there are also alternative sorting methods where the content of the keys are used to obtain their position without need to compare them to each other. We will call them methods "content based" in this work. They can obtain better results because real machines allow many other operations besides comparison [AHNR98]. Examples of content based methods are *radixsort* [Knu73] and *groupsort* [BSA97].

In this work, we are interested in developing sorting algorithms for special cases of lists of integers that follow skewed distributions (we will call these lists *skewed lists of integers*). In this case, previous knowledge about the lists to be sorted can be used to reduce the sorting times by using special purpose algorithms (note that the lists is not partially sorted, and therefore the main core of adaptive sorting algorithms [ECW92] does not apply). We present here a new special purpose content based sorting algorithm which deals efficiently with these lists, taking linear average time to sort them. We also show an example application of our algorithm in text compression, using it to sort the list of frequencies of words in natural language texts. These lists follow the Zipf's distribution [Zip49], a well known skewed distribution.

2 Groupsort algorithm

The *groupsort* algorithm [BSA97] partitions the range of numbers to be sorted in K groups, called *buckets*. It makes a first pass over the list to compute the number of elements in each group. After this pass, the elements are distributed on their buckets according to these values. Figure 1 shows an example of *groupsort* working on a list of 16 elements and using 3 buckets ($K = 3$) with ranges 1–30, 31–60 and 61–90. A first pass over the list indicates that the buckets 1, 2 and 3 have 7, 5 and 4 elements respectively. The algorithm distributes the elements on the buckets as shown in the lower part of Figure 1. After this stage, each partition is sorted again individually as a new list. The authors have suggested that this new sorting can be done with the *groupsort* itself or another sorting algorithm according to an efficiency criterion defined by them.

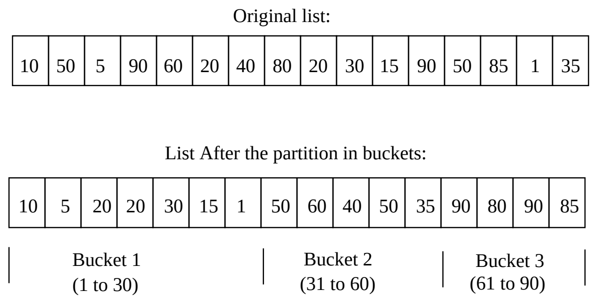

Figure 1. Example showing the partition of a list in *groupsort* using 3 buckets ($K = 3$)

The performance of *groupsort* depends on how uniform is the distribution of the elements on the buckets. The authors have suggested that the subrange value of each bucket should be chosen so that the elements are evenly distributed across the buckets. In practice, this restriction decreases the performance due to the additional cost to calculate the subranges.

3 Remainingsort strategy

We are interested in designing a sorting algorithm for skewed lists of integers. A common feature in these lists is that most elements have small values, and the number of elements with a given value x quickly decreases as x increases. ¿From these observations, we derive a sorting strategy based on *groupsort*. The main modification we have proposed is to divide the list in $K + 1$ buckets, where the first K buckets have range 1 (i.e. they accept only one value) and the last bucket gets the remaining numbers of the list. Therefore, the first K buckets are sorted on the partition step and there is only one remaining bucket to sort after that — we will call this bucket "remaining bucket". This is an important improvement when our strategy is compared against *groupsort*, where all the K buckets should be sorted after the partition step. We will call this new sorting strategy *remainingsort*.

Figure 2 shows an example where we divide the list using four buckets. The first three buckets have range 1, getting the values 1,2 and 3 respectively. The remaining elements are placed on the fourth bucket (the "remaining bucket"), which is the only one we need to sort.

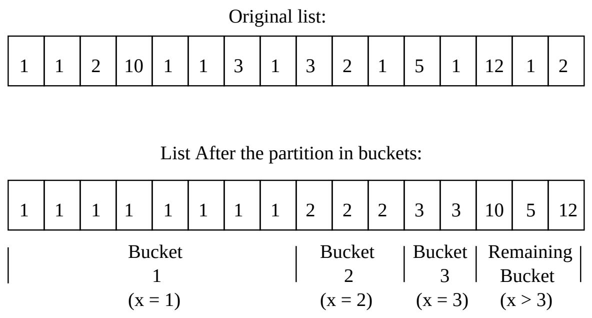

Figure 2. Example showing the partition of a list using 3 buckets of range 1 ($K = 3$) plus the remaining bucket

If K is sufficiently large the remaining bucket can be sorted with any conventional algorithm without changing the overall complexity of the sorting process. On the other hand, the value of K should be small in order to reduce the extra space used by the algorithm (which is K counters). Therefore, a good choice is to establish a K that gives an $O(n/\log n)$ remaining bucket size, where n is the number of elements in the list. This choice allows an $O(n)$ time sorting of the remaining bucket. Given a list L with n elements to be sorted, and a function G such that the sequence $G(1), ..., G(n)$ corresponds the list L sorted, then a good value for K would be $G(\lfloor n/\log n \rfloor)$. This value can be obtained through a linear time algorithm to obtain the k-th element from the unordered array (for $k = n/\log n$) or it can be estimated directly if the G distribution is known.

Our sorting algorithm uses therefore $K = \lceil G(\lfloor n/\log n \rfloor) \rceil$ extra space to sort the list. The total time is that of initializing the K counters, performing a linear pass over the list to increment the counters, making another pass over the counters to generate the elements in order, and sorting the remaining bucket. Since we have selected K as the minimum value that makes the final pass linear, the total cost is $O(K+n) = O(G(n/\log n)+n)$.

If K is too large, the extra space required by the algorithm will not be practical, and another sorting strategy should be used. In particular, the algorithm has overall linear time if $G(n/\log n) = O(n)$. For skewed distributions, $G(i)$ tends to decrease quickly as i increases, and therefore it is more probable that this condition holds.

The complete *remainingsort* algorithm follows.

1. Compute $K = \lceil G(\lfloor n/\log n \rfloor)\rceil$ either by estimation of G or by a linear time algorithm for the k-th element.
2. Create $K+1$ counters for the number of elements on each bucket, where the $(K+1)$-th is the "remaining bucket".
3. Count the number of elements on each bucket by a linear pass over the list. Each element $x \leq K$ increments counter x, otherwise it increments counter $K+1$.
4. Put the elements in their corresponding buckets.
5. Sort the remaining bucket using a conventional sorting procedure

4 Sorting Zipfian Sequences with Remainingsort

The general idea described in the last section can be applied with good results to a wide variety of list of integers that follows skewed distributions. An important example is given when G follows the Zipf's distribution [Zip49]. The Zipf's law states that, if we order the n elements of the list in decreasing order (obtaining $x_1, ..., x_n$), then the value of the first element is i^θ times that of the i-th element, for every i, for a constant θ. This means that the value of the i-th element is $x_i = N/(i^\theta H)$, where $N = \sum_{i=1}^{n} x_i$, $H = H_n^{(\theta)} = \sum_{j=1}^{n} 1/j^\theta$, and θ is a small constant value greater than 1.

We show now that if $K = O(x_n(\log n)^\theta)$ and the list to be sorted follows the Zipf's law, then the number of elements in the remaining bucket is $O(n/\log n)$. From the Zip's law, the value of the element at position $n/\log n$ of the list in decreasing order is:

$$K = x_{(\frac{n}{\log n})} = \frac{N}{(\frac{n}{\log n})^\theta H} \frac{(\log n)^\theta N/n^\theta}{H} \qquad (1)$$

We show now that $N/(x_n n^\theta) = H$. Since the smallest element in the list is x_n, we can use Zipf's law to write its value as $N/(n^\theta H)$. Equating both expressions we have

$$x_n = \frac{N}{n^\theta H} \quad \Rightarrow \quad H = \frac{N}{x_n n^\theta} \qquad (2)$$

and therefore, from Eq. (1) and Eq. (2) we have

$$K = O(x_n(\log n)^\theta).$$

We can sort the remaining bucket in $O(n)$ time using a conventional comparison based sorting algorithm. This is because its size is $n' = O(n/\log n)$, and a classical sort on it costs $O(n' \log n') = O(n)$. Therefore, the overall time complexity of the *remainingsort* algorithm is $O(n)$ as well. The extra space used to perform the sorting is only the necessary to compute the size of each bucket, which is $O(x_n(\log n)^\theta)$. It is important to observe that x_n tends to be a small number due to the characteristics of the Zipfian distributions. If it is not possible to estimate a reasonable value of x_n before the sorting, it can be obtained also in linear time without change the average time complexity. However, we are interested in the more general $K = c(\ln n)^\theta$.

5 An Example of Application

We present now an application of the *remainingsort* to reduce the time to construct Huffman codes [Huf52] when the alphabet symbols are words and the source to be compressed is a natural language text. This coding scheme, known as word-based Huffman [BSTW86], has important applications on information retrieval systems, were it is used to reduce the storage costs and to improve the search performance [ZM95, MNZBY98b, MNZBY98a]. In fact, the Huffman code construction represents only a small portion of the overall compression times. However, we are investigating alternative schemes to allow editing in compressed text where the Huffman code is rebuilt periodically. Contributions to reduce the Huffman coding construction times can be decisive to the success these new ideas.

More formally, a word-based Huffman code can be defined as a minimum-redundancy code. Given a source alphabet $S = [s_1, ..., s_n]$, where each symbol s_i has an associated weight (or probability) p_i, a minimum-redundancy code C of base b is a list $[c_1, ..., c_n]$, where $c_i \in \{0, ..., b-1\}^*$ and such that C is prefix free (which means c_i is not a prefix of c_j $\forall$ $i \neq j$) and $\sum_{i=1}^{n} p_i |c_i|$ is minimized. It is usual to denote minimum-redundancy codes as Huffman codes due to a famous algorithm proposed by David Huffman [Huf52] to solve this problem.

Some recent works have presented fast algorithms to construct Huffman codes [MK95, MT98, MPL98].

However, these works make the assumption that the alphabet list is given in increasing order of symbol frequencies (or weights). Therefore, it is necessary to sort the alphabet list before applying these algorithms. Furthermore, the Huffman code construction phase is linear, while sorting the alphabet list can be $O(n \log n)$ using general comparison based algorithms. Hence, sorting the frequencies is the heaviest part of the algorithm.

The alphabet used when constructing word-based Huffman codes is composed of words extracted from a natural language text. It is widely accepted in the information retrieval community that the frequency distribution of these words follows the Zipf's law [Zip49], where N is the total number of words in the text and n is the size of the vocabulary. Therefore, the *remainingsort* algorithm for Zipfian distributions can be applied in the sorting phase of the word-based Huffman code construction. The combination of the algorithm presented in [MK95] with our new sorting algorithm results in a fast linear time method to construct word-based Huffman codes.

Experiments with natural language texts show that the value of the constant θ for natural language texts is between 1.5 and 2.0 [ANZ97]. Further, the least frequent word of a text (x_n) has a small number of occurrences that is close to 1 (in almost all natural language the texts there are many words with frequency 1 [BYN97]). Therefore, the extra space used by the *remainingsort* algorithm in this application is $K = O((\log n)^2)$, which is a small extra space requirement.

To show the usefulness of the idea we made experiments using literary texts from the TREC collection [Har95]. We have chosen the following texts: *ap* Newswire (1989), *doe* - Short abstracts from *doe* publications, *fr* - Federal Register (1989), *wsj* - Wall Street Journal (1987, 1988, 1989) and *ziff* - articles from *Computer Selected* disks (Ziff-Davis Publishing). We put all these files together to obtain a text vocabulary composed of 681 thousand words. We have also produced fragments of this vocabulary by parsing the TREC files and storing partial vocabularies from size $1,000$ to $681,000$. All the experiments were run on a SUN SparcStation 4 with 96 megabytes of RAM running Solaris 2.5.1.

The first objective of the experiments was to determine a good practical value for the constant c. c should be large enough to reduce the time necessary to sort the remaining bucket, and should be as small as possible in order to reduce the extra space and counter processing time used by the algorithm. Figure 3 shows experiments with a large range of values for the constant c when the *remainingsort* algorithm is applied to the TREC vocabulary. The figure shows that the best result is obtained with the value $c = 6$. After this point, the time to sort the remaining bucket is not significant anymore and the running time is determined by the time to divide the elements in their buckets, so increasing c will increase the running time.

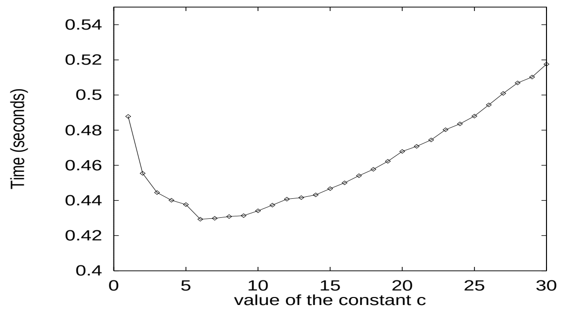

Figure 3. Sorting times for the *remainingsort* algorithm when varying the constant c from 1 to 30, running over the whole TREC vocabulary

After determining a good value for the constant c, we made experiments comparing the performance of the *remainingsort* algorithm against an adaptive quicksort specially designed to deal with lists with a large amount of equal keys [Weg85, ECW92], which we will reference as *quicksort-equal*. The idea used in this quicksort is to not process sublists where all the elements have the same value. We have considered other alternatives to compare with, such as all the general sorting algorithms described in [Knu73] and also the adaptive algorithms described in [ECW92, PM95]. However, the faster algorithm we found to compare with *remainingsort* when sorting text vocabularies by frequency is the quicksort variation presented in the experiments.

Figure 4 shows the performance of these algorithms when running over the TREC vocabulary. Our algorithm in these experiments was more then twice faster than *quicksort-equal*.

Table 1 shows the best fit curves obtained when applying the least squares method to the data presented in Figure 4. We have matched the time results with the best curves $C_1 n(\ln(n))^{\alpha_1}$ and $C_2 n^{\alpha_2}$, where C_1, α_1, C_2 and α_2 are constants. This table indicates that the running times of *remainingsort* increase at the same ratio of the input, which matches our analytical results about the linearity of the algorithm when sorting vocabulary frequencies. In this experiments, the *quicksort-equal* algorithm has resulted in a sublinear curve, but the best practical time results where obtained by the *remainingsort* and the curves are so close that this difference tends to be almost constant.

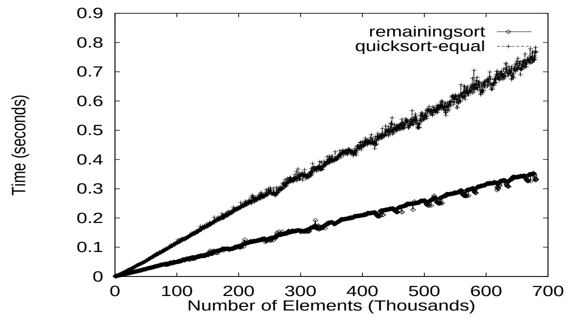

Figure 4. Sorting times for the *remainingsort* and *quicksort-equal* when constructing the Huffman code for the TREC vocabulary

Method	$n(\log(n))^{\alpha_1}$		n^{α_2}	
	α_1	error	α_2	error
remainingsort	0.000	2.10%	1.000	2.10%
quicksort-equal	-0.012	3.25%	0.990	3.22%

Table 1. Best values for α and errors when fitting the curves $n(\log(n))^{\alpha}$ and n^{α} with the *remainingsort,* and *quicksort-equal*

6 Conclusions

We have presented a special purpose technique to sort lists that follow skewed distributions. This is a simple idea which can be applied to a wide variety of situations, but its usefulness depends on the contents of the lists to be sorted. We have also used this more general idea to derive an algorithm to sort lists that follows the Zipf's distribution. We have shown analytically that this algorithm has linear average time and needs $O(x_n(\log n)^{\theta})$ extra space, where x_n is the smallest element in the list and n is the number of elements in the list. We also have shown an application where this algorithm is used to fast sorting alphabets when building word-based Huffman codes on natural language texts in linear average time.

The general ideas presented here can be used in many real world situations. We have shown in this article just an example. Another example of application we are considering now is to use the remainingsort strategy to rank documents in information retrieval system. The idea is to apply the *remainingsort* when the documents are ranked by the number of links that points to them. As shown in [BP98], a list of documents, when they are represented by this number, follows a skewed distribution where the most popular element is the value 1, which is a good situation for the *remainingsort* strategy.

Acknowledgements

We wish to acknowledge the helpful comments of the anonymous referees, who give important suggestion that were used in the final version of this article.

References

[AHNR98] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *J. Computer & Systems Science*, pages 74–93, 1998.

[ANZ97] M. D. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In R. Baeza-Yates, editor, *Proc. of the Fourth South American Workshop on String Processing*, volume 8, pages 2–20. Carleton University Press International Informatics Series, 1997.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh World-Wide Web Conference*, 1998.

[BSA97] A. Burnetas, D. Solow, and R. Agrawal. An analysis and implementation of an efficient in-place bucket sort. *Acta Informatica*, 34:687–700, 1997.

[BSTW86] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.

[BYN97] R. Baeza-Yates and G. Navarro. Block addressing indices for approximate text retrieval. In *Proc. of Sixth ACM International Conference on Information and*

Knowledge Management (CIKM'97), pages 1–8, Las Vegas, Nevada, 1997.

[ECW92] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–475, December 1992.

[Har95] D. K. Harman. Overview of the third text retrieval conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology Special Publication.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 1(5):10–15, 1962.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. of the Institute of Electrical and Radio Engineers*, volume 40, pages 1090–1101, 1952.

[Knu73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

[MK95] A. Moffat and J. Katajainen. In-place calcula-
tion of minimum-redundancy codes. In S.G. Akl, F. Dehne, and J.-R. Sack, editors, *Proc. Workshop on Algorithms and Data Structures*, pages 393–402. LNCS 955, Springer-Verlag, 1995.

[MNZBY98a] E. S. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *SPIRE'98*, pages 90–95, Santa Cruz de la Sierra, Bolivia, September 1998. IEEE Computer Society.

[MNZBY98b] E. S. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. of the ACM Sigir'98*, pages 298–306. York Press, August 1998.

[MPL98] R. Milidiu, A. Pessoa, and E. Laber. Inplace, simple, and fast length-restricted prefix coding. In *SPIRE'98*, pages 50–59, Santa Cruz de la Sierra, Bolivia, September 1998. IEEE Computer Society.

[MT98] A. Moffat and A. Turpin. Efficient construction of minimum-redundancy codes for large alphabets. *IEEE Transactions on Information Theory*, 44(4):1650–1657, July 1998.

[PM95] O. Petersson and A. Moffat. A framework for adaptive sorting. *Discret Applied Mathematics*, 59:153–179, 1995.

[She59] D. L. Shell. A high speed sorting procedure. *Communications of ACM*, 2(7):30–32, 1959.

[Weg85] L. M. Wegner. Quicksort for equal keys. *IEEE Transactions on Computers*, 34(4):362–367, April 1985.

[Zip49] G. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, 1949.

[ZM95] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. *Software Practice and Experience*, 25(8):891–903, 1995.