In Proc. 35th Annual IEEE Symposium on Foundations of Computer Science, pages 714-721. IEEE Computer Society Press, 1994.

A New Efficient Radix Sort

Arne Andersson

Department of Computer Science
Lund University
Box 118, S-221 00 Lund, Sweden
email: arne@dna.lth.se

Stefan Nilsson

Department of Computer Science
Lund University
Box 118, S-221 00 Lund, Sweden
email: stefan@dna.lth.se

Abstract

We present new improved algorithms for the sorting problem. The algorithms are not only efficient but also clear and simple. First, we introduce *Forward Radix Sort* which combines the advantages of traditional left-to-right and right-to-left radix sort in a simple manner. We argue that this algorithm will work very well in practice. Adding a preprocessing step, we obtain an algorithm with attractive theoretical properties. For example, n binary strings can be sorted in $\Theta\left(n \log\left(\frac{B}{n \log n} + 2\right)\right)$ time, where B is the minimum number of bits that have to be inspected to distinguish the strings. This is an improvement over the previously best known result by Paige and Tarjan. The complexity may also be expressed in terms of H, the entropy of the input: n strings from a stationary ergodic process can be sorted in $\Theta\left(n \log\left(\frac{1}{H} + 1\right)\right)$ time, an improvement over the result recently presented by Chen and Reif.

Keywords: algorithms, sorting, radix sort, Forward Radix Sort, entropy.

1 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.

In this article, we study this more general approach in the context of sorting. Specifically, we study the problem of arranging binary strings in lexicographic order. A new approach to this problem was recently presented by Chen and Reif [4]. They introduced a randomized sorting algorithm for binary strings from a *stationary ergodic process* (see Section 3.2) with entropy H, yielding an expected cost

$$\Theta\left(n \log\left(\frac{\log n}{H} + 2\right)\right).$$

This was claimed to be the first algorithm to adapt to an unknown input distribution.

We take a more general approach based on *distinguishing prefixes.* Let B denote the total length of all distinguishing prefixes, i.e. the minimum number of bits that have to be examined in order to sort the strings. Let $\bar{B} = B/n$. For a stationary ergodic process satisfying a certain *mixing condition* (see Section 3.2) the following is true [17]:

$$\lim_{n\to\infty} \frac{E(\bar{B})}{\log n} = \frac{1}{H}.$$

Hence, for any algorithm where the cost can be expressed in terms of $\bar{B}$, the cost can also be expressed in terms of H. However, the reverse implication is not true. Therefore, stating the complexity of an algorithm in terms of $\bar{B}$, making no assumptions about how the input is generated, is more general than using the entropy of a stationary ergodic process.

Distribution-sensitive algorithms that are analyzed in terms of $\bar{B}$, and which therefore adapt to unknown distributions, have been presented before. One example is the sorting algorithm based on partition refinement by Paige and Tarjan [16]. For this algorithm, the cost for sorting n binary strings is

$$\Theta\left(n\left(\frac{\bar{B}}{\log n} + 1\right)\right).$$

2 Results

We present improved algorithms for the sorting problem that are both simple and efficient. The basic algorithm, *Forward Radix Sort*, combines the advantages of traditional left-to-right and right-to-left radix sort. We also give an extended algorithm that consists of a preprocessing step and an integer sorting step.

The analysis is made in terms of $\bar{B}$, the average number of bits in a distinguishing prefix, and w, the number of bits in a machine word. We also show how to apply the results to input from a stationary ergodic process.

The basic algorithm has attractive features from a practical point of view.

- It is simple.
- Under the assumptions about real world input made by Chen and Reif [4], the basic algorithm runs in $\Theta(n)$ time, while their rather complicated algorithm runs in $\Theta(n \log\log n)$ time.
- Experimental investigations further indicate the favorable behavior.

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.

Applying different integer sorting algorithms as subroutines and choosing an optimal trade-off between preprocessing and sorting, we get the following results:

- In the simplest case, when bucketing is used as a subroutine, the complexity is the same as that achieved by Paige and Tarjan [16],
$$\Theta\left(n\left(\frac{\bar{B}}{\log n}+1\right)\right).$$
- Using the sorting algorithm by Kirkpatrick and Reisch [12] we get two cases. Assuming that $w = \Omega(\bar{B})$, the cost is
$$\Theta\left(n\log\left(\frac{\bar{B}}{\log n}+2\right)\right).$$
For input from a stationary ergodic process, the complexity is
$$\Theta\left(n\log\left(\frac{1}{H}+1\right)\right).$$
In general, making no assumption on the size of w, the cost is
$$O\left(n\left(\frac{\bar{B}}{w}+1+\log\frac{w}{\log n}\right)\right).$$
These costs improve over both the result by Paige and Tarjan [16], and that of Chen and Reif [4].
- Using the integer sorting algorithm by Albers and Hagerup [1], the cost is
$$\Theta\left(n\sqrt{\bar{B}\,\frac{\log n\log\log n}{w}}\right),$$
or
$$\Theta\left(n\log n\sqrt{\frac{\log\log n}{Hw}}\right).$$
Comparing this with the algorithm above, we see that this algorithm has a better asymptotic time complexity if
$$w=\Omega\left(\bar{B}\,\frac{\log n\log\log n}{\log^2\frac{\bar{B}}{\log n}}\right).$$
In particular, the algorithm runs in linear time if $w = \Omega(\bar{B}\log n\log\log n)$.

3 Preliminaries

3.1 Notation and model of computation

We follow the notation of Mehlhorn [15]. Let $\Sigma = \{1, 2, \ldots, m\}$ be an alphabet, with the standard arithmetic linear order. We consider strings $x^1, \ldots, x^n$ over Σ. Denote the length of x^i by l_i and write $x^i = x^i_1 x^i_2 \cdots x^i_{l_i}$. A lexicographic ordering is defined in the usual way. Let $x = x_1 \cdots x_k$ and $y = y_1 \cdots y_l$. Then x is smaller than y in the lexicographic ordering if there is an i, $0 \le i \le k$, such that $x_j = y_j$ for $1 \le j \le i$ and either $i = k < l$ or $i < k$, $i < l$ and $x_{i+1} < y_{i+1}$.

Consider the distinguishing prefixes of the strings, i.e. the shortest prefixes of $x^1, \ldots, x^n$ that are pairwise different. The distinguishing prefix of a string x^i that is a prefix of one of the other strings is defined to be the entire string x^i. Denote the length of the distinguishing prefix of x^i by s_i. The number s_i can also be characterized as the depth of x^i in the m-ary trie formed by the strings. Let $S = \sum_{i=1}^{n} s_i$ and let $\bar{S} = S/n$. S is the total number of characters that must be inspected in order to arrange the strings in lexicographic order, and $\bar{S}$ is the average length of a distinguishing prefix.

In particular, we will study binary strings. In this case it is natural to use an alphabet of size 2^b, since most machines can extract and manipulate short bit strings efficiently. To be more precise, we will consider a unit cost random access machine, RAM, with word length w. We assume that $b = O(w)$, so that operations can be performed in constant time on binary strings of length b. Furthermore, we assume that $n \leq 2^w$, since otherwise n will be larger than the available address space of the machine. Since we need to simultaneously discuss both the number of distinguishing bits and the number of distinguishing characters in an alphabet of size 2^b, it is convenient to introduce the special notation B and $\bar{B}$ to denote the total and average number of distinguishing bits, respectively, while S and $\bar{S}$ denote the number of distinguishing b-bit characters.

3.2 Statistical model and entropy

How the average number of distinguishing bits $\bar{B}$ depends on the distribution of the input data is a well studied topic. In fact, $\bar{B}$ equals the average depth of a leaf in the binary trie formed by the strings, and the statistical behavior of trie structures is very well understood. We consider input from stochastic processes $\{X_i\}$ with values from an alphabet Σ. In particular, we study *stationary ergodic processes* [2], the traditional statistical model used in information theory. For such a process, one can define the *entropy*

$$H = \lim_{n \to \infty} -\frac{1}{n} E(\log P\{X_i = x_i; 1 \leq i \leq n, x_i \in \Sigma\}).$$

Informally, H can be viewed as a measure of disorder. If the strings have a high degree of disorder and hence high entropy, we can expect to be able to differentiate between two strings by only looking at a small number of characters. On the other hand, if the entropy is low the strings will have a large number of characters in common and hence it will take more computational power to tell them apart. This idea has been formalized by Pittel [17]. To state the theorem of Pittel we need to introduce a mixing condition:

> Denote F_a^b the σ-field generated by $X_a, \ldots, X_b$, $1 \leq a \leq b$; there exist two positive constants $c_1 \leq c_2$ and an integer $b_0 > 0$ such that for all $1 \leq a \leq a + b_0 \leq b$,
>
> $$c_1 P(A)P(B) \leq P(AB) \leq c_2 P(A)P(B)$$
>
> whenever $A \in F_1^a$, $B \in F_{a+b_0}^b$.

For independent binary strings from a stationary ergodic process obeying this condition,

$$\lim_{n \to \infty} \frac{E(\bar{B})}{\log n} = \frac{1}{H}. \qquad (1)$$

4 Forward Radix Sort

Traditionally, radix sort algorithms fall into two major categories, those who process the strings forward, from left to right, and those who process the strings backward, from right to left. The forward scanning algorithm has been referred to as radix-exchange sort [14, 19], top-down radix sort [10], and MSD radix sort [11, 13]. Similarly, the backward scanning algorithm is called straight radix sort, bottom-up radix sort, LSD radix sort, or just radix sort. These algorithms are well known and we merely give a short description in order to point out the major differences.

forward scan Split the strings into groups according to their first character and arrange the groups in sorted order. Apply the algorithm recursively on each group separately, with the first character removed. Groups containing only one string need not be processed further. After the ith step of the algorithm, the input strings will be properly sorted according to their first i characters.

backward scan Split the strings into groups according to their last character and arrange the groups in sorted order. Apply the algorithm recursively on *all* strings, with the last character removed. After the ith step of the algorithm, the input strings will be properly sorted according to their last i characters.

These 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.

Here we present a version of radix sort that combines the advantages of forward and backward scanning. The algorithm maintains the invariant that after the ith pass, the strings are sorted according to the first i characters. The sorting is performed by separating the strings into groups. Initially, all strings are contained in the same group, denoted group 1. This group will be split into smaller groups, and after the ith pass all strings with the same first i characters will belong to the same group. The groups are kept in sorted order according to the prefixes seen so far. 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. The ith step of the algorithm is performed in the following way:

1. Traverse the unfinished groups in sorted order and insert each string x, tagged by its current group number, into bucket number x_i (recall that x_i is the ith character in x).

2. Traverse the buckets in sorted order and put the strings back into their respective groups in the order as they occur within the buckets.

3. Traverse the groups separately. If the kth string in group g differs from its predecessor in the ith character, split the group at this string. The new group is numbered $g + k - 1$.

Observe that when moving strings, we do not move the strings themselves but pointers to them. In this way, each string movement is guaranteed to take constant time. One can also observe that Step 1 and 2 may be replaced by another sorting algorithm. (This will be done in Section 5.1.3.)

The algorithm can be implemented in many ways. The buckets can be implemented as an array of m linked lists. To keep track of the groups we can use an array of size n, where each entry contains one string. The groups are stored consecutively and in sorted order in this array. Also, two pointers are associated with each group. One pointer indicates the start of the next unfinished group. The other pointer is used to indicate the point of insertion in the group during collection of strings in Step 2. Using this data structure, we can split a group in constant time. Also, the unfinished groups can be traversed in time proportional to the total number of strings in these groups.

To see that the algorithm is correct, we observe that after Step 2, the strings within a group will be sorted according to their ith character. Since the groups are sorted according to their first $i - 1$ characters, the strings will be sorted according to their first i characters.

4.1 Discussion

Forward Radix Sort runs in $\Theta(S + n + m \cdot s_{\max})$ time, where $s_{\max}$ is the length of the longest distinguishing prefix. The last term comes from the fact that the algorithm runs in $s_{\max}$ passes, and in each pass m buckets are visited. In the later passes, the number of remaining strings can be considerably smaller than m and the cost of visiting empty buckets may be significant. A simple way to decrease the cost is to switch to a standard comparison based sorting algorithm when the number of strings that remains to be sorted is small.

As an example, consider the following method for sorting a set of binary strings. Let a character consist of $b = \left\lceil \frac{\log n}{2} \right\rceil$ bits. Then $m = \Theta(\sqrt{n})$ and $S = \Theta\left(\frac{B}{\log n} + n\right)$. Run the basic algorithm, using this alphabet size, but stop when the number of strings remaining in the unfinished groups is less than $\sqrt{n}$. The remaining strings are sorted with a standard comparison-based algorithm. During each pass, the number of characters examined is at least $\sqrt{n}$ and therefore the cost of traversing the buckets does not affect the asymptotic cost. The total number of characters examined will be $O\left(\frac{B}{\log n} + n\right)$. Hence, the total time complexity of this sorting method is $O\left(\frac{B}{\log n} + n + \text{cost of sorting } \sqrt{n} \text{ elements}\right)$. Under the reasonable assumption that the cost of sorting the $\sqrt{n}$ remaining elements is small compared to the first part of the algorithm, the cost will be the same as for the algorithm by Paige and Tarjan [16]. However, our algorithm is much simpler.

In practice, this algorithm compares favorably with that of Chen and Reif [4]. They point out that in real world applications, the value of $1/H$ will be a small constant. Furthermore, their analysis requires that a subset of v strings can be sorted in $O(v \log v)$ time. Based upon these assumptions, they claim that their algorithm runs in $\Theta(n \log \log n)$ time in practice. However, under these assumptions, the complexity of the algorithm sketched above is $\Theta(n)$.

The favorable properties of Forward Radix Sort are further emphasized by experimental results. Chen and Reif [4] made experiments using input with a compression rate between 2 and 4. Extrapolating from these results, they claimed that their algorithm would beat the UNIX system quicksort routine when the number of elements exceeds $32,000,000$. We have performed experiments on data with an even larger variation in compression rate, from 1 (no compression possible) to 5. A relatively simple implementation of Forward Radix Sort turned out to be considerably faster than highly tuned versions of quicksort in almost all cases. In fact, the break-even point was below 1000.

5 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.

I. Create P.

II. Sort P.

III. Run the basic algorithm using P to avoid looking at empty buckets.

Step I The strings are processed from left to right in passes, dividing them into groups. This time, however, the groups will not occur in sorted order. We only maintain a weaker invariant: after the ith pass the strings in an unfinished group will have the same first i characters. In detail, the following actions are performed in the ith pass.

1. Traverse the unfinished groups and insert each remaining string x, tagged by its current group number, into bucket number x_i. Also, maintain a list of nonempty buckets; when a string is added to an empty bucket, the bucket is added to this list.
2. Traverse the list of nonempty buckets and put the strings back into their respective groups in the order as they occur within the buckets.
3. Traverse the groups separately. If the kth string in group g differs from its predecessor in the ith character, split the group at this string. The new group is numbered $g+k-1$. Also, every character c that participates in the splitting of a group is added to P as a pair (i, c).

Observe that we store a pair (i, c) only if the character c is used to split a group in pass i. As a result P contains at most $2n-2$ elements.

Step II We observe that the pairs are already sorted according to their first coordinate, the pass number, since the elements were collected in this order. To sort the pairs we need therefore only to sort according to the second coordinate and then collect the strings into groups as indicated by the first coordinate.

This simple approach will always work, but will be inefficient if the first coordinate attains large values. However, it is always possible to collect efficiently. Instead of using the number of the pass, i, we assign a number i' to the pairs in such a way that pairs with the same first coordinate are assigned the same number:

$$\begin{aligned} i'_1 &= i_1 \\ i'_k &= \begin{cases} i'_{k-1} & \text{if } i_k = i_{k-1} \\ i'_{k-1}+1 & \text{otherwise} \end{cases} \end{aligned}$$

Using these numbers, collection can be done in linear time and space, since each i' is less than n.

Step III Run the basic algorithm, using the information in P. This time we do not need to visit empty buckets and hence the running time of this step will be $\Theta(S+n)$.

5.1 Discussion

The extended algorithm sorts n strings in $\Theta(S+n+$ cost of sorting n characters) time. For binary strings, treating b bits as a character, we have an alphabet of size 2^b. Recall that we assume that $b = O(w)$, so that operations can be performed in constant time on binary strings of length b. The number of significant characters $S = \Theta(B/b)$ and hence the extended algorithm runs in time

$$\Theta\left(\frac{B}{b} + n + T\left(n, 2^b, w\right)\right), \tag{2}$$

where $T(n, 2^b, w)$ is the time to sort n integers in the range $[0, 2^b - 1]$ on a unit cost RAM with word length w. The amount of extra space required is $\Theta(n + 2^b)$. This can be reduced to $O(n)$ using universal hashing [3, 5], yielding a randomized algorithm with the same expected asymptotic time complexity.

There is a trivial lower bound: since the entire distinguishing prefixes must be read at least once and each element must be processed at least once, the minimum cost is

$$\Omega\left(\frac{B}{w}+n\right). \tag{3}$$

In particular, choosing $b = w$ in Expression 2, we observe that the running time equals the lower bound plus the cost of sorting n integers of length w. In effect, the string sorting problem has been reduced to an integer sorting problem.

In the next three sections we investigate how the choice of alphabet size and integer sorting algorithm affects the behavior of the extended algorithm.

5.1.1 Applying bucket sort

As a very straightforward application, we note that an algorithm with behavior similar to the sorting algorithm by Paige and Tarjan [16] can be obtained by using plain bucket sort. Choosing $b = \lceil \log n \rceil$, we may sort the list of pairs by distributing them among $2^b = \Theta(n)$ buckets. In this case, $T(n, 2^b, w) = \Theta(n)$ and the total time complexity is

$$\Theta\left(n\left(\frac{\bar{B}}{\log n}+1\right)\right). \tag{4}$$

For input from a stationary ergodic process we get the bound

$$\Theta\left(\frac{n}{H}\right). \tag{5}$$

According to the trivial lower bound above, this algorithm is optimal if the word length is $\Theta(\log n)$.

5.1.2 Applying the algorithm by Kirkpatrick and Reisch

Under the assumption that $b \le w$, the sorting algorithm by Kirkpatrick and Reisch [12] sorts n integers in the range $[0, 2^b - 1]$ in $\Theta(n \log(b/\log n + 2))$ time. The algorithm uses $\Theta(2^{b/2})$ extra space, but this can be reduced to $O(n)$ with universal hashing [3, 5]. Using this sorting algorithm as a subroutine we get the time complexity

$$\Theta\left(n\left(\frac{\bar{B}}{b}+1+\log\left(\frac{b}{\log n}+2\right)\right)\right). \tag{6}$$

It is not hard to see that the minimum of this expression is $\Theta(n \log(\bar{B}/\log n + 2))$. However, it is not a priori clear how to choose b to achieve this minimum, since $\bar{B}$ cannot be expected to be known in advance. Below we show how to get around this problem.

Start by choosing $b = \lceil \log n \rceil$ and run Step I of the extended algorithm until either all strings have been separated or the number of b-bit characters processed during the step exceeds $2n$. If not all strings have been separated in this first step, start over again but this time choose $b = 2\lceil \log n \rceil$. Continue doubling b until Step I terminates after having processed less than $2n$ characters. If this process stops after one step, $B = O(n)$ and the algorithm will run in linear time. If more than one step is needed, $bn \le B \le 2bn$ and hence $b = \Theta(\bar{B})$. If $w = \Omega(\bar{B})$ each of these preprocessing steps will take $O(n)$ time. Thus, the total time for this doubling procedure will be $\Theta(n \log(\bar{B}/\log n + 2))$. Next, we run Step II using the value of b for which step I was completed. This requires $\Theta(n \log(\bar{B}/\log n + 2))$ time as well. Hence, the cost of the doubling can be included in the sorting cost without affecting the asymptotic complexity. In total, for word length $w = \Omega(\bar{B})$, we obtain the time complexity

$$\Theta\left(n\log\left(\frac{\bar{B}}{\log n}+2\right)\right), \tag{7}$$

an improvement over the bucketing algorithm.

The time bound above is valid only for $w = \Omega(\bar{B})$. But even if $w = o(\bar{B})$ the algorithm of Kirkpatrick and Reisch can be used. We add one more condition to the doubling step: the doubling is terminated when $b > w$ and we run the rest of the algorithm with $b = w$. This yields the following time complexity:

$$\Theta\left(n\left(\frac{\bar{B}}{w}+1+\log\frac{w}{\log n}\right)\right). \tag{8}$$

Relating this to the lower bound in Expression 3 we see that the algorithm is optimal if

$$\bar{B} = \Omega(w(1+\log\frac{w}{\log n})).$$

In order to compare the result with the bucketing algorithm, we make the additional assumption that $\log n = o(w)$, since otherwise the bucketing algorithm is optimal (see Section 5.1.1). Now, it is easy to see that Expression 8 is less than Expression 4: $\bar{B}/w = o(\bar{B}/\log n)$ since $\log n = o(w)$, and $\log(w/\log n) = o(\bar{B}/\log n)$ since $w = o(\bar{B})$. In fact, as soon as the bucket algorithm is suboptimal this algorithm is better.

Combining Expression 7 and Expression 1 we see that for input from an ergodic process with entropy H, the complexity becomes

$$\Theta\left(n\log\left(\frac{1}{H}+2\right)\right). \tag{9}$$

This is a clear improvement over the

$$\Theta\left(n \log\left(\frac{\log n}{H} + 2\right)\right) \qquad (10)$$

expected time achieved by Chen and Reif [4]. The two results can be directly compared since they employ the same statistical model. (In their article, the mixing condition from Section 3.2 is not mentioned, but it is implicitly assumed since their proof depends on an application of a theorem by Szpankowski [20] that requires this very same condition.)

The assumption that $w = \Omega(\bar{B})$ is needed in the analysis made by Chen and Reif. Although they claim to handle input strings of arbitrary length, their result relies on the assumption that each distinguishing prefix fits into a constant number of machine words [18]. For example, they assume that element comparisons can be made in constant time. Furthermore, if $w = o(\bar{B})$ it is not always possible to achieve the time bound in Expression 10, as can be seen by the following simple computation. Consider the case where $1/H = \Theta(\log n)$ and $w = O(\log n)$. ¿From Equation 1 it follows that $w = o(\bar{B})$. But Equation 1 and Expression 3 gives a lower bound of $\Omega(n \log n)$, contradicting the upper bound $O(n \log\log n)$ of Expression 10.

5.1.3 Applying the algorithm by Albers and Hagerup

The sorting algorithm by Albers and Hagerup [1] sorts n integers in the range $[0, 2^b - 1]$ in $\Theta(n)$ time on a random access machine with word length $w = \Omega(b \log n \log\log n)$. Let $c = \Theta(w/(\log n \log\log n))$, then we can sort n integers containing c bits in $\Theta(n)$ time. For our purpose we need an algorithm that can handle longer integers and therefore we make an extension of their algorithm.

To sort n integers of length $b = \Omega(c)$, we run Forward Radix Sort with the following modifications. We use c-bit characters and in Step 1, instead of inserting the strings into n buckets, we sort them in $\Theta(n)$ time using the algorithm by Albers and Hagerup. In Step 2, instead of traversing the buckets, we traverse the sorted list. Sorting n b-bit integers in this way takes $O(n(1 + b/c))$ time. (A version of traditional right-to-left radix sort modified in a similar way could also have been used.)

Using this modified algorithm as a subroutine, the cost of the extended algorithm becomes

$$\Theta\left(n\left(\frac{\bar{B}}{b} + \frac{b}{c}\right)\right).$$

We minimize this expression by choosing $b = \left\lceil \sqrt{c\bar{B}} \right\rceil$. This choice of b yields the time complexity

$$\Theta\left(\sqrt{\frac{\bar{B}n}{c}}\right) = \Theta\left(n\sqrt{\bar{B}\,\frac{\log n \log\log n}{w}}\right). \qquad (11)$$

Once again, we use the doubling technique from Section 5.1.2 to choose b. Start with $b = c$ bits and continue doubling b until Step I can be finished after inspecting at most $2n$ characters. The cost of this procedure is smaller than the total cost, since

$$n \log\frac{\bar{B}}{c} = O\left(\sqrt{\frac{\bar{B}n}{c}}\right).$$

For input from a stationary ergodic process with entropy H, we obtain a cost of

$$\Theta\left(n \log n \sqrt{\frac{\log\log n}{Hw}}\right). \qquad (12)$$

Comparing Expression 11 with Expression 7, we see that this algorithm has a better asymptotic time complexity if

$$w = \Omega\left(\bar{B}\,\frac{\log n \log\log n}{\log^2 \frac{\bar{B}}{\log n}}\right). \qquad (13)$$

In particular, the algorithm runs in linear time if $w = \Omega(\bar{B} \log n \log\log n)$.

Acknowledgements

We would like to thank Kerstin Andersson, Ola Petersson, and Kurt Swanson for valuable comments.

References

[1] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. In *Proc. 3rd ACM-SIAM SODA*, pages 463–472, 1992.

[2] P. Billingsley. *Ergodic theory and information.* John Wiley & Sons, 1965.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[4] S. Chen and J. H. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and convex hull on entropy bounded inputs. In *Proc. 34th IEEE FOCS*, pages 104–112, 1993.

[5] Th. H. Cormen, Ch. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* McGraw-Hill, 1990.

[6] L. Devroye. *Lecture Notes on Bucket Algorithms.* Birkhäuser, 1985. ISBN 0-8176-3328-6.

[7] W. Dobosiewicz. Sorting by distributive partitioning. *Information Processing Letters*, 7(1), 1978.

[8] E. H. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[9] G. H. Gonnet. *Interpolation and Interpolation-Hash Searching.* Ph.D. Thesis, University of Waterloo, Canada, 1977.

[10] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1991.

[11] J. H. Kingston. *Algorithms and data structures: design, correctness, analysis.* Addison-Wesley, 1990. ISBN 0-201-41705-7.

[12] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28:263–276, 1984.

[13] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Massachusetts, 1973.

[14] U. Manber. *Introduction to Algorithms.* Addison-Wesley, 1989. ISBN 0-201-12037-2.

[15] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, 1984. ISBN 3-540-13302-X.

[16] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[17] B. Pittel. Asymptotical growth of a class of random trees. *The Annals of Probability*, 13(2):414–427, 1985.

[18] J. H. Reif, 1994. Personal communication.

[19] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, Massachusetts, 1988. ISBN 0-201-06673-4.

[20] W. Szpankowski. (Un)expected behavior of typical suffix trees. In *Proc. of 3rd SODA*, pages 422–431, 1992.

[21] D. E. Willard. Searching unindexed and nonuniformly generated files in loglog n time. *SIAM Journal on Computing*, 14(4), 1985.

[22] A. C.-C. Yao and F. F. Yao. The complexity of searching an ordered random table. In *Proc. IEEE FOCS*, 1976.