PREPRINT. The ACM Journal of Experimental Algorithmics. Volume 3, Article 7, 1998.

Implementing Radixsort

Arne Andersson
Lund University
and
Stefan Nilsson
Helsinki University of Technology

We present and evaluate several optimization and implementation techniques for string sorting. In particular, we study a recently published radix sorting algorithm, Forward radixsort, that has a provably good worst-case behavior. Our experimental results indicate that radix sorting is considerably faster (often more than twice as fast) than comparison-based sorting methods. This is true even for small input sequences. We also show that it is possible to implement a radixsort with good worst-case running time without sacrificing average-case performance. Our implementations are competitive with the best previously published string sorting programs.

Categories and Subject Descriptors: F.2.2 [**Theory of Computation**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Sorting

Additional Key Words and Phrases: String sorting, radix sorting, adaptive radixsort, forward radixsort

---

1. INTRODUCTION

Radix sorting is a simple and very efficient sorting method that has received too little attention. A common misconception is that a radix sorting algorithm either has to inspect all the characters of the input or use an inordinate amount of extra time or space. However, with a careful implementation efficient implementations are possible as has been shown by several researchers [Davis 1992; McIlroy et al. 1993]. In fact, radix sorting can be implemented to run considerably faster than strictly comparison-based sorting for typical data. In this article we show that it is possible to implement radix sorting in such a way as to guarantee also good worst-case behavior. This is achieved by an algorithm, Forward radixsort [Andersson and Nilsson 1994], that combines the advantages of traditional LSD and MSD radixsort.

Before going into details we formally state the problem and our theoretical model of computation. Let $\Sigma = \{1, 2, \ldots, m\}$ be an alphabet of characters, with the standard arithmetic linear order. We consider strings $x^1, \ldots, x^n$ over Σ. Denote

---

To appear in Journal of Experimental Algorithmics.
Name: Arne Andersson
Affiliation: Department of Computer Science, Lund University
Address: Box 118, SE-22100 Lund, Sweden
Name: Stefan Nilsson
Affiliation: Department of Computer Science, Helsinki University of Technology
Address: P. O. Box 1100, FIN-02014 HUT, Finland

the length of x^i by l_i and write $x^i = x^i_1 x^i_2 \cdots x^i_{l_i}$. An alphabetic order 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 alphabetic order 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 $x_{i+1} < y_{i+1}$. The problem is to arrange a given collection of strings in alphabetic order.

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 [Mehlhorn 1984], 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 $n \le 2^w$, since otherwise n will be larger than the available address space of the machine. In our experiments, the strings are binary strings from the ASCII character set and the word length is 32. However, many sorting problems can be cast in this form. The basic data abstractions of most computers are integers, floating-point numbers (typically defined by the IEEE 754 standard), and character strings. All of these are well suited for radix sorting.

Both LSD and MSD radixsort are based on *Bucketsort*. Consider the problem of sorting a set of integers drawn from a small set. Bucketsort maintains one bucket for each possible key value, puts the elements in their respective buckets, and collects the elements during a traversal of the buckets.

LSD *radixsort* sorts the strings in several stages, using bucketsort at each stage, starting with the least significant character (LSD stands for "Least Significant Digit"). This means that the algorithm has to inspect all characters of the input. Hence we cannot expect optimal performance from this algorithm and therefore we do not discuss it any further.

Another, perhaps more natural alternative, is to scan the input starting with the most significant character. This algorithm is known as MSD *radixsort*. The algorithm works as follows.

—Split the strings into groups according to their first character and arrange the groups in ascending order.

—Apply the algorithm recursively on each group separately, disregarding the first character of each string. Groups containing only one string need no further processing.

The advantage of this algorithm is that it inspects only the *distinguishing prefixes*, the minimum number of characters that has to be inspected to assure that the strings are, in fact, sorted.

Definition 1. The *distinguishing prefixes* of a set of strings $x^1, \ldots, x^n$ are defined as 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.

The main problem with MSD radixsort is that it might inspect a large number of empty buckets. The two traditional techniques to overcome this problem are to switch to a simple comparison-based sorting algorithm, such as Insertion sort, for small subproblems and to use a heuristic, such as keeping track of the minimum and maximum character that is encountered, to reduce the number of buckets that needs to be inspected. This typically works well for text files in ASCII format. Another recently proposed solution [Bentley and Sedgewick 1997], is to replace bucket sort by a comparison-based method. The algorithm is presented as a variation of quicksort, where the string comparisons have been replaced by appropriate character comparisons. This is, however, just a question of different perspectives. One might equally well view the algorithm as a variation of MSD radixsort with bucketing replaced by quicksort.

We take a different approach and study two algorithms: *Adaptive radixsort* and *Forward radixsort*. Adaptive radixsort is a simple modification of plain MSD radixsort where the size of the alphabet is chosen dynamically. This algorithm turns out to be very good in practice. Forward radixsort combines the advantages of LSD and MSD radixsort: It scans the input starting with the most significant character, but performs bucketing only once per character position. Forward radixsort has a provably good worst-case behavior and it is almost as fast as Adaptive radixsort.

2. IMPLEMENTING MSD RADIXSORT

To implement radix sorting algorithms efficiently you need to be careful. In this section we discuss some basic implementation issues that pertain to most types of radix sorting algorithms.

The main design decision is whether to represent the input by a linked list or an array. The array representation has the advantage that the permutation can be performed in place, but we do not get a stable algorithm. Algorithms based on linked lists have the advantage of being simple to implement. Earlier experiments [McIlroy et al. 1993] indicate that the list-based implementations are faster on some architectures and array-based implementations on others. Since our primary goal is to compare different algorithms and optimization schemes and not to write the fastest possible sorting program for a particular architecture we have chosen to implement all of our algorithms using linked lists.

An important observation is that to implement a string sorting algorithm efficiently we should not move the strings themselves but only pointers to them. In this way, each string movement is guaranteed to take constant time. This is not a major restriction. In fact, it is common to store the characters of each string in consecutive memory locations and represent each string 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.

Switching to a simple comparison-based method (typically Insertion sort) for small subproblems is a well-known technique to improve the speed of sorting algorithms. This technique turns out to be very important in a practical implementation and it is used in all the programs discussed.

When implementing MSD radixsort we want to avoid using a new bucket table for each invocation of bucketsort. This can be arranged if we use an explicit stack to keep track of the flow of computation [McIlroy et al. 1993]. The last bucket is treated first and the other sublists to be sorted are pushed on the stack. This is illustrated in Figure 1. The sublists on the stack that are waiting to be sorted are suggested by the dotted line.

One possible problem with this implementation is that the stack can be sizable. It is not difficult to construct an example where the stack will contain one entry for

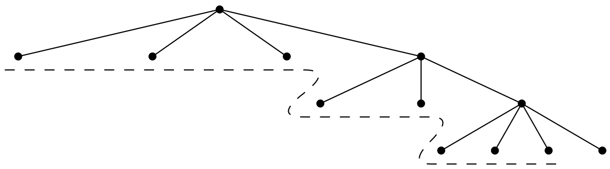

Fig. 1. A snapshot of MSD radixsort.

each key. We suggest a simple way to overcome this problem. The idea is to take advantage of the fact that short sublists, containing at most k elements, are sorted using a comparison-based algorithm. This sorting is performed before the lists are pushed onto the stack. This means that if both the list on the top of the stack and the next list to be pushed onto the stack contain at most k elements we do not need to allocate a new stack record. Both lists are already sorted and we can simply append the new list to the end of the list on top of the stack. In this way the stack will never contain two consecutive entries that both have less than k elements. Hence, the total size of the stack will be at most $2n/k$. In practice, however, the stack will typically be much smaller. Not only does this simple optimization give a considerable reduction of the stack size, it is also likely to improve the running time.

When choosing the size of the alphabet we are faced with a fundamental problem: A large alphabet reduces the total number of passes but increases the total number of buckets that must be inspected. In practice one can sometimes use heuristic methods to be able to use larger alphabets. For example, if the elements are ASCII-encoded strings from the English language they typically only contain characters in the range 32 to 126. A simple heuristic [McIlroy et al. 1993] to avoid inspecting many empty buckets is to record the minimum and maximum character encountered during the bucketing phase and then only look at the buckets in between these extreme values.

A certain amount of clustering is often present in many kinds of data. One way to take advantage of this fact is to treat consecutive elements with a common character as a single unit when moving them into a bucket. Since we are using a linked-list representation it is possible to move this sublist of identical elements in constant time. This simple optimization improves the performance in many practical situations.

3. ADAPTIVE RADIXSORT

In this section we present a modified version of MSD radixsort. In this algorithm we adapt an old idea, which has been used primarily for sorting real numbers. The resulting algorithm is very simple, has a good expected running time for a large class of distributions, and works very well in practice.

Sorting based on *distributive partitioning* [Dobosiewicz 1978] is an old idea. The algorithm sorts n real numbers by distributing them into n intervals of equal width. The process is repeated recursively for each interval that contains more than 1 element: The numbers are distributed into a number of intervals that equals the number of keys. By combining this approach with a Quicksort partitioning step using the median as pivot element Dobosiewicz gets a sorting algorithm with $O(n)$ expected time for uniform data and $O(n \log n)$ worst-case time.

Ehrlich also presents several searching and sorting algorithms based on the idea of distributive partitioning [Ehrlich 1981]. In particular, he defines a new data structure, the *N-tree*. At the root the N-tree partitions the key-space into n intervals of equal width, where n is the number of keys; the same technique is applied recursively on the children.

The approach of distributive partitioning can of course also be applied to radix sorting algorithms. A natural extension of MSD radixsort is to choose the size of the alphabet adaptively: The alphabet size is chosen as a function of the number of elements remaining. For example, for binary strings a natural choice is to use characters consisting of $\log k$ bits to distribute the elements of a group of size k into $\Theta(k)$ buckets. In this way the number of buckets that need to be traversed will be proportional to the number of inspected characters. Using this scheme the cost of visiting buckets matches the cost of inspecting bit patters from the strings. However, in a worst-case scenario the elements will be split into many small groups already after a few steps and the algorithm will read only a small constant number of bits at a time. Hence, the worst-case time complexity is $\Theta(n + B)$, where B is the total number of bits of the distinguishing characters.

Observe that the total number of bits read by the algorithm might be larger than for plain MSD radixsort, since we might read a number of bits beyond the distinguishing prefix. However, this happens at most once for each string and hence will account for at most a linear extra cost. In practice this extra cost is typically overshadowed by the gain from not having to inspect as many superfluous empty buckets.

There is a close relationship between N-trees and Adaptive radixsort. The algorithm performs at most $O(n)$ work at each level of the corresponding N-tree. The expected average depth of an N-tree is approximately 1.8 for uniformly distributed data [Tamminen 1983] and hence Adaptive radixsort runs in linear expected time for this kind of data. Although this give some indication of the algorithm's behavior, random uniform data is not a very good model of real data. In text strings, for example, duplicate keys or long repetitions are likely to occur. For a more detailed average case analysis of bucket algorithms see [Devroye 1985; Tamminen 1985].

Our implementation of Adaptive radixsort uses two different alphabet sizes: 8 bits and 16 bits. The details of the implementation are the same as for standard MSD radixsort. However, for larger alphabets it is useful to apply a more sophisticated heuristic to avoid looking at too many empty buckets. We have used a simple heuristic. Two consecutive 8-bit characters are used for bucketing, yielding an alphabet of size 65 536. The idea is to keep track of the characters that occur in the first and second position. For example, assuming that n_1 and n_2 different characters have been found in the first and second position, respectively, we only need to inspect $n_1 n_2$ buckets. For ASCII-text this number is typically much smaller than the total 65 536 buckets.

4. FORWARD RADIXSORT

Several authors [Cormen et al. 1990; Kingston 1990; Mehlhorn 1984] have pointed out that MSD radixsort has a bad worst-case performance due to fragmentation of the data into many small sublists. In this section we present a simple MSD radixsort algorithm, *Forward radixsort* [Andersson and Nilsson 1994], that overcomes this problem. The algorithm combines the advantages of LSD and MSD radixsort. The main strength of LSD radixsort is that it inspects a complete horizontal strip 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. Forward radixsort starts with the most significant character, performs bucketing only once for each horizontal strip, and inspects only the significant characters.

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 pass of the algorithm is performed in the following way.

—Traverse the groups that are not yet 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. If all inspected characters are equal, no bucketing is performed.

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

—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 $g + k - 1$.

4.1 Implementation

To keep track of the groups we use a linked list, where each entry contains a list of elements with common prefixes. Each group also has a pointer that indicates the start of the next unfinished group. Using this data structure we can split a group in constant time. The unfinished groups can be traversed in time proportional to the total number of strings in these groups. Figures 2 and 4 give a snapshot of the group list before and after the third pass.

The buckets are implemented by an array of linked lists, one for each character in the alphabet. An important observation is that it is not necessary to use an explicit tag for each element. Instead each bucket contains lists of elements with the common tag stored in the head of the list. Figure 3 illustrates this. A new bucket entry is created only if the tag of the entry differs from the tag of the list that is to be put into the bucket. Note that the algorithm will always insert an element into the first list of the bucket and hence the insertion time is constant. In combination with the usual strategy of switching to a simple comparison-based sorting algorithm for small groups this gives a considerable space reduction. The

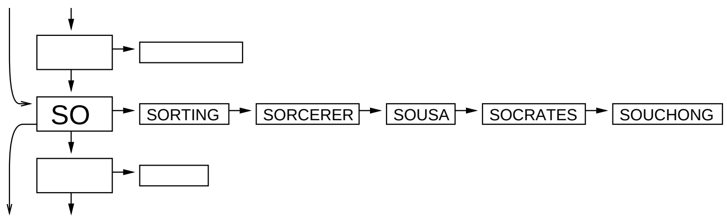

Fig. 2. Snapshot of the group list after two passes.

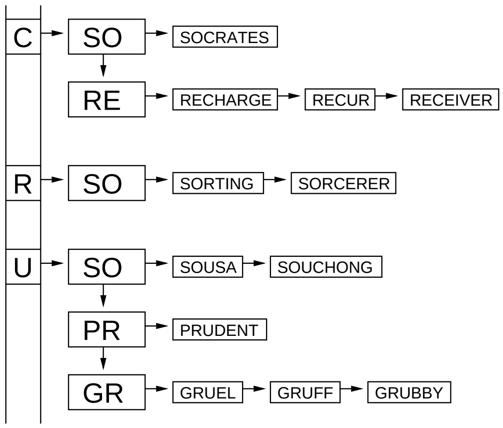

Fig. 3. Snapshot of the buckets after the insertion of all remaining strings with prefix "SO."

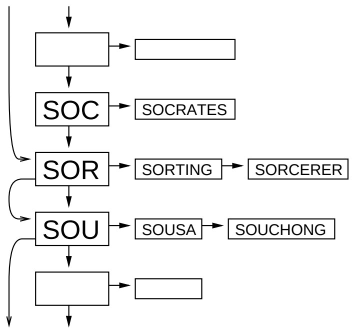

Fig. 4. Snapshot of the group list after three passes.

reason is that we do not need to store any extra tag information with each element, but only at the head of the lists and the number of headers is typically much smaller than the number of elements. Also note that the memory needed for these headers may be reused after each pass.

We have implemented the algorithm with two different alphabets: 8-bit and 16-bit characters. For the 16-bit alphabet we have used the same heuristics as for Adaptive radixsort to avoid inspecting too many empty buckets. Finally, the algorithm checks, in each pass, whether all inspected characters are equal. If this is the case, no bucketing is performed.

4.2 Discussion

Forward radixsort runs in $O(S + n + m \cdot S_{\max})$ time, where S is the total number of characters of the distinguishing prefixes, $S_{\max}$ 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. (The term n is needed since also empty strings must be inspected one time.) The last term comes from the fact that the algorithm runs in $S_{\max}$ passes and visits m buckets in each pass.

The worst-case running time is also bounded by $O(S + n + m^2)$. 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 unfinished 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 visisted is $O(m^2)$ in this case.

Note that these bounds hold also if we switch to Insertion sort for short subsequences. We just need to be careful to implement the comparison function in such a way that it only inspects characters belonging to the distinguishing prefixes. This is not hard: A standard comparison function that inspects the strings from left to right and stops when it encounters the first mismatch fulfills this requirement.

In summary, we have the following theorem.

THEOREM 1. *A collection of n strings from an alphabet of size m can be sorted in*

$$O(S + n + m \cdot \min(m, S_{\max}))$$

time, where S is the total number of distinguishing characters and $S_{\max}$ is the length of the longest distinguishing prefix.

As an application, consider the following method for sorting n binary strings. We regard the strings as consisting of characters with $1/2 \cdot \log n$ bits. Then the size of the alphabet $m = \Theta(\sqrt{n})$. Furthermore, the number of distinguishing characters $S = \Theta(B/\log n + n)$ and from Theorem 1 we immediately get the following corollary.

COROLLARY 1. *A sequence of n strings from a binary alphabet can be sorted in $O(B/\log n + n)$ time, where B is the total number of distinguishing bits.*

The main virtue of Forward radixsort is its simplicity. The time bound of Corollary 1 was also derived by Paige and Tarjan using a different and more complicated algorithm [Paige and Tarjan 1987]. Their algorithm first constructs an explicit unordered path-compressed trie structure. During this construction the edges of the trie are put into buckets according to the characters associated with the edges. During the second phase of the algorithm these buckets are traversed and the outgoing edges of each internal node of the trie are arranged in ascending order. The sorted list is produced during a preorder traversal of the trie.

As shown in the original article about Forward radixsort [Andersson and Nilsson 1994] it is possible to improve this asymptotic running time. In fact, it is possible to reduce the problem of sorting n strings to an integer sorting problem. This reduction can be performed in $O(S+n)$ time. This algorithm is more complicated and requires that we traverse the distinguishing prefixes of the strings two times. Therefore we do not expect it to be competitive for most practical applications and we have chosen not to include it in this empirical study.

The time bound of Theorem 1 seems to indicate that a 16-bit alphabet would be too large for small to medium problem sizes (the quadratic term is 2^{32}). A simple solution would be to use a 16-bit alphabet only for the first passes of the algorithm, when the number of strings remaining is large, and then to switch to an 8-bit alphabet. In practice this is not necessary. The combination of the 16-bit character heuristic and switching to Insertion sort for subproblems of less than 30 elements works to our advantage. To see this note that inspecting empty buckets is a cheap operation: The loop is tight and contains only a simple test. Furthermore, the memory locality is good, since the buckets are stored in consecutive memory positions. In fact, we can allow the algorithms to inspect as many as 10 empty buckets per string, without any major slow-down of the algorithm. If the number of remaining elements is larger than a few thousand, we therefore don't have to worry about the cost of inspecting empty buckets. On the other hand, if only a few thousand elements remain, it might happen that we have to inspect many empty buckets. But if this happens, we can be sure that the size of the problem has been substantially reduced. To see this, note that the 16-bit character heuristic implies that if the algorithm inspects many buckets there will be many different characters in the input. Furthermore, if there are less than 30 strings containing a particular character, these strings will be sorted immediately using Insertion sort.

5. EXPERIMENTAL RESULTS

There is only one way to know which sorting algorithm is the fastest in practice. You have to implement the algorithm and measure the running time. This may seem like a simple task, but there are many pitfalls. The results will be influenced by such factors as compilers, operating systems, and machine architecture. Also, the amount of work spent on implementing a particular algorithm is likely to affect its efficiency. We have implemented the algorithms in a similar format with linked lists and we have refrained from micro-optimizations such as loop unrolling and manual register allocation. Such optimizations are typically better performed by a compiler than a human being.

We present time measurements for MSD radixsort, Adaptive radixsort, and Forward radixsort with 8-bit and 16-bit alphabets. All of these algorithms have been implemented in a consistent way using linked lists. Furthermore, we make comparisons with competitive versions of quicksort [Bentley and McIlroy 1993], radixsort [McIlroy et al. 1993], and the recently introduced Multikey quicksort [Bentley and Sedgewick 1997]. These results are harder to interpret since these implementations use an array data structure and some of them apply micro-optimizations. Still, it is clear that our algorithms are competitive.

5.1 Method

The code development has been done on a SPARC station ELC using the SUN `cc` compiler, the GNU `gcc` compiler, the `gdb` debugger, and the `tcov` line-count profiler. The measurements presented in this paper were produced on a SPARC station 20 with 160 megabytes of internal memory with the `gcc` compiler. Measurements for several other machines are available. In all cases we have used the highest optimization level. We have measured the running time in seconds using the C `clock()` function. The standard deviation was low, but we did encounter occasional outliers. These atypical timings were, of course, always slower than the typical value. To be able to perform measurements on small problem instances we created multiple copies of the input and measured the total time for sorting each of them in turn.

It is common to study the behavior of sorting algorithms for uniformly distributed data. But realistic sorting problems are usually far from random. Also, uniformly distributed keys can be sorted very efficiently using a simple bucket scheme. To get more realistic timings we have chosen to use samples of real text. Experiments were performed on a collection of standard texts from the Calgary/Canterbury text compression corpus [Bell et al. 1990]. We used all text files within this corpus and each line of the input constitutes one key. The average length of a line is about 40 characters in the texts. Since all of these files are relatively small we have also used a number of freely availably non-copyrighted texts from Project Gutenberg.[1]

It is much harder to find realistic worst-case data. To inspect the complete data space is of course unfeasible and we can never be completely sure that a particular input really gives rise to the worst possible performance. However, as an illustration we have generated badly fragmented data in an attempt to make the radix sorting algorithms behave badly. The idea is to force the algorithm to immediately split the strings into a large number of groups. These groups contain both small and large characters so that all buckets will have to be examined in each phase of the algorithm. Furthermore, the groups are constructed in such a way that only one element is eliminated during each bucket phase. Also, the groups are larger than the Insertion sort breakpoint. In Figure 5 we see the first few lines of such a text from an alphabet $\{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}\}$. In this particular example the Insertion sort breakpoint is 3. For the experiments we have used the breakpoint 16 and an alphabet of size 255.

The following algorithms were examined.

FRS 8. Forward radixsort using 8-bit characters. The minimum and maximum character is recorded in each bucket phase and Insertion sort is used to sort small groups.

FRS 16. Similar to the previous algorithm, but with 16-bit characters. To avoid

[1] `http://gutenberg.etext.org/`

AAZ
AAAZ
AAAAZ
AAAAAZ
AAAAAAZ
AAAAAAAZ
AAAAAAAA
AAAAAAAA
AAAAAAAA
ABZ
ABAZ
ABAAZ
ABAAAZ
ABAAAAZ
ABAAAAAZ
ABAAAAAA
ABAAAAAA
ABAAAAAA

Fig. 5. Fragmented data.

inspecting all buckets, the algorithm uses the 16-bit character heuristic discussed in Section 3.

MSD. MSD radixsort using 8-bit characters. The minimum and maximum character is recorded. The algorithm moves sublists of elements with a common character as a unit. An explicit stack is used to keep track of the flow of computation. In this way we only need to use one bucket array. The algorithm switches to Insertion sort for small groups.

Adaptive. The algorithm is identical to the one above except that it uses two different character sizes, 8 bits and 16 bits. In the 8-bit case it keeps track of the minimum and maximum character in each bucket phase. In the 16-bit case it uses the same heuristic as FRS 16.

In addition to these algorithms we have also studied several previously published sorting algorithms.

Quicksort. Quicksort has a long history of implementation and optimization. The version engineered by Bentley and McIlroy seems to be the clear winner [Bentley and McIlroy 1993]. The algorithm is constructed to work for all kinds of input and hence has some overhead. We therefore used a stripped-down version that is specifically tailored for character strings.

Multikey. Multikey quicksort is a recent algorithm [Bentley and Sedgewick 1997]. It may be viewed as an implementation of MSD radixsort using Quicksort instead of bucketing. The algorithm uses the same optimizations as the quicksort algorithm above.

Program C. An array based implementation of MSD radixsort [McIlroy et al. 1993]. The article contains three additional implementations, but this one is typically the fastest. The algorithm uses a fixed 8-bit alphabet and performs the permutations in place. It keeps track of the minimum and maximum character in each bucket phase. The algorithm switches to Insertion sort for small subproblems.

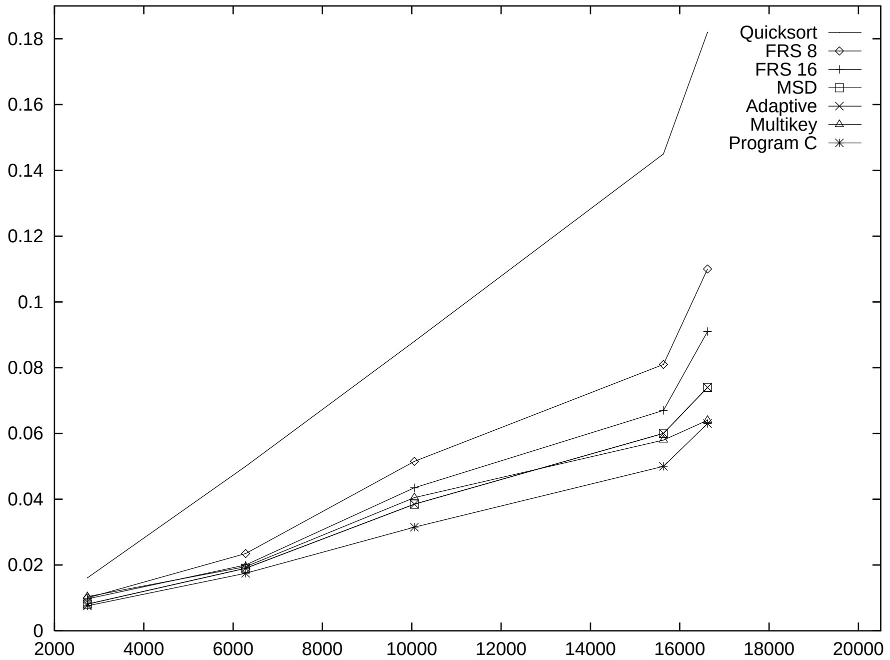

Fig. 6. String sorting. Middle-sized files. The running time in seconds. (Note that the connecting lines in the figures are not meant to imply any correlation between files of different sizes. They merely connect different measurements for one algorithm.)

5.2 Discussion

Several researchers [Davis 1992; McIlroy et al. 1993] have independently found that a carefully coded radix sorting algorithm outperform comparison-based algorithms by a large margin. In fact, our experimental data indicates that this is true already for small problem instances consisting of only a few hundred keys. Some typical timing measurements are shown in Figure 6. These results were produced on a SPARC station 20 with the `gcc` compiler. We obtained similar results also on other machines. Observe that the differences between the different radixsort implementations are small. In particular, Adaptive radixsort performs indistinguishably from plain MSD radixsort for small files: Adaptive radixsort uses the larger 16-bit alphabet only for subproblems of more than 1500 strings. For Forward radixsort a larger alphabet size yields an improvement already for about 5000 elements.

Figure 7 shows timings for some larger files. As expected, Adaptive radixsort is faster than MSD radixsort and the difference between FRS 16 and FRS 8 becomes more noticeable.

Figure 8 shows the running time for the fragmented data described above. We see that Forward radixsort is the most robust algorithm. Even in this “worst-case” setting a straightforward radix sorting algorithm runs only slightly slower than Quicksort.

In the figures we have also included the fastest sorting algorithms that we have

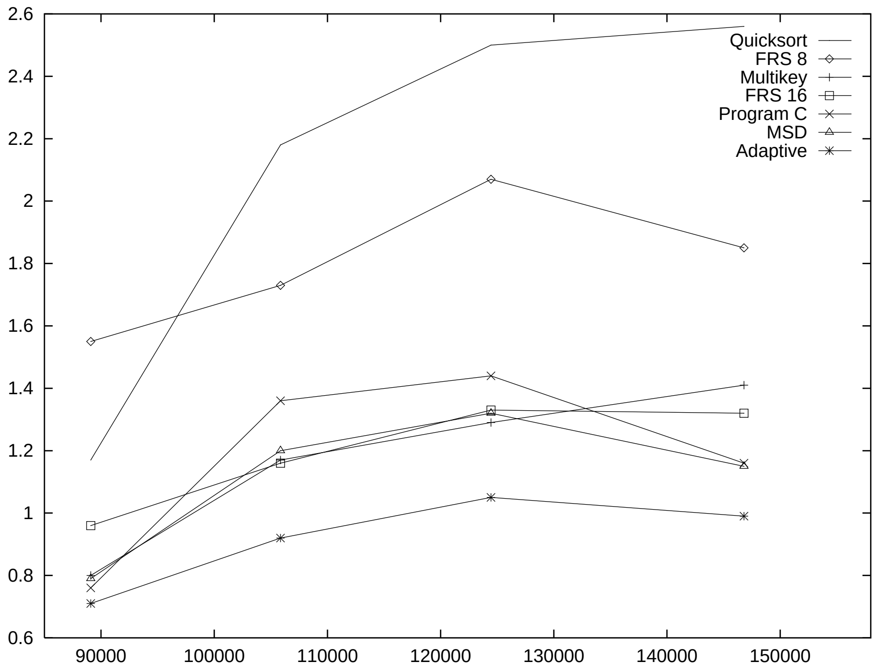

Fig. 7. String sorting. Large files. The running time in seconds.

been able to find in the literature. These time measurements are harder to evaluate since the implementations use different levels of optimizations. In particular, all of these algorithms use an array to represent the data. However, the time differences between array and linked-list implementations seem to be small. We are not able to announce one algorithm as the winner, but it is clear that radix sorting algorithms are faster than comparison-based algorithms. In fact, the quicksort [Bentley and McIlroy 1993] used in our experiments is considerably faster than any other comparison-based method that we have investigated and yet the radix sorting algorithms are about twice as fast as this algorithm.

One advantage of Forward radixsort is that it reduces the amount of bucketing. In many implementations the cost associated with each bucket operation will be larger than in our simple variant. The alphabetic order of the characters might be different from the numerical order and there may be several character codes mapping to the same bucket. For example, the characters 'E', 'e', and 'é' are typically treated as equivalent for the purpose of alphabetic sorting. For the 16-bit Unicode character set [The Unicode Consortium 1991] these factors can be important and Forward radixsort might well be the most feasible radix sorting algorithm in this case.

Multikey quicksort [Bentley and Sedgewick 1997] is also a strong contender. On average it is very fast and, as can be seen from Figure 8, it is unaffected by our "worst-case" data. However, it should be noted that this experiment doesn't exhibit the worst-case behavior of this algorithm. Multikey quicksort uses quicksort instead of bucketsort to perform the character sorting required in each stage of the algo-

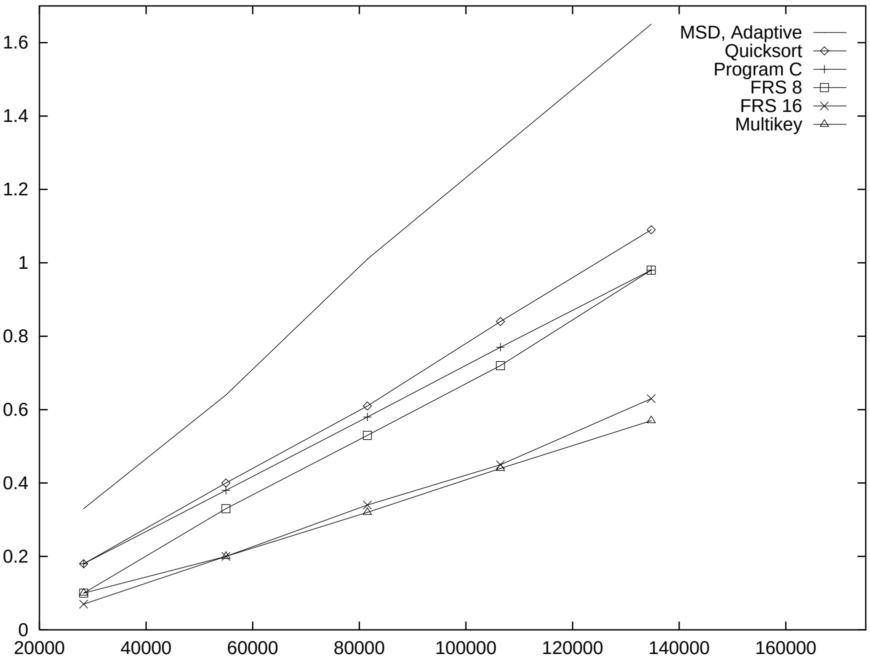

Fig. 8. String sorting. Fragmented data. The running time in seconds.

rithm. The speed of quicksort compared to bucketsort depends on both the size of the alphabet and the number of characters to be sorted. A careful implementation of quicksort sorts k characters from an alphabet of size m in $\Theta(k \log(\min(m, k)))$ worst-case time, while bucketing runs in $\Theta(k + m)$ time. We have not considered data that exhibits the worst-case behavior of Multikey quicksort.

5.3 Lessons Learned

The single most important optimization for the sorting algorithms discussed in this paper is to switch to a simple comparison-based algorithm when only a small number of elements remain to be sorted. Previous research [Bentley and McIlroy 1993] indicate that Insertion sort is best suited. The exact value of the breaking point is not crucial. Values in the range 10 to 30 give satisfactory results. For Forward radixsort this optimization reduces the running time by about 40% and for Adaptive radixsort the reduction is almost 50%.

Since a large part of the total running time is spent inside the Insertion sort routine it is important to implement this routine carefully. We have taken advantage of the fact that in MSD radix sorting algorithms the strings to be compared are known to have a common prefix and hence there is no need to compare the strings starting at the first position. For highly repetitious data this optimization reduces the overall running time by as much as 10%.

One of the biggest problems to overcome in the implementation of Forward radixsort was the large space overhead. However, we found a simple and very efficient solution to this problem: 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%.

But we can do even better 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. In fact, the total number of group records that are allocated is typically only 6% of the total number of elements. This optimization also reduces the running time with another 4%. The same technique was also used to minimize the explicit run-time stack used by Adaptive radixsort.

With these two optimizations little extra space is required. In our implementation a group record consists of 6 words and a bucket record consists of 5 words. In addition to the space needed to represent the pointers of the linked list, Forward radixsort uses less than 0.8 extra words per element for the data in our experiments.

There are many design decisions to be made when implementing a radix sorting algorithm, such as how to choose the alphabet size. A theoretically attractive scheme is to choose the number of bits proportional to the logarithm of the remaining number of elements. Another approach is to restrict the alphabet to certain fixed sizes. On a 32-bit machine it is natural to extract a number of bits that is a multiple of 8. In this way we avoid potential problems with word alignment. A priori it is not clear which of these approaches is the best. In fact, preliminary experiments indicated that for Forward radixsort the free-choice method was best and for Adaptive radixsort 8-bit alignment was slightly better, but the differences were small. To keep the code simple we chose to implement our algorithms with word alignment.

6. CONCLUSIONS

We have investigated the performance of a number of string sorting algorithms. It is evident that radix sorting algorithms are much faster than the more frequently used comparison-based algorithms. On the average Adaptive radixsort was the fastest algorithm. Forward radixsort was only slightly slower and it has a guaranteed good worst-case behavior. Forward radixsort is also well suited for large alphabets such as the Unicode 16-bit character set [The Unicode Consortium 1991].

REFERENCES

ANDERSSON, A. AND NILSSON, S. 1994. A new efficient radix sort. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science* (1994), pp. 714–721.

BELL, T. C., CLEARY, J. G., AND WITTEN, I. H. 1990. *Text Compression.* Prentice Hall.

BENTLEY, J. L. AND MCILROY, M. D. 1993. Engineering a sort function. *Software—Practice and Experience 23*, 11, 1249–1265.

BENTLEY, J. L. AND SEDGEWICK, R. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of the Eight Annual ACM-SIAM Symposium on Discrete Algorithms* (1997), pp. 360–369.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms.* McGraw-Hill.

DAVIS, I. J. 1992. A fast radix sort. *The Computer Journal 35*, 6, 636–642.

DEVROYE, L. 1985. *Lecture Notes on Bucket Algorithms.* Birkhäuser.

DOBOSIEWICZ, W. 1978. Sorting by distributive partitioning. *Information Processing Letters 7*, 1, 1–6.

Ehrlich, G. 1981. Searching and sorting real numbers. *Journal of Algorithms 2*, 1, 1–12.

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

McIlroy, P. M., Bostic, K., and McIlroy, M. D. 1993. Engineering radix sort. *Computing Systems 6*, 1, 5–27.

Mehlhorn, K. 1984. *Sorting and Searching*, Volume 1 of *Data Structures and Algorithms.* Springer-Verlag.

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

Tamminen, M. 1983. Analysis of N-trees. *Information Processing Letters 16*, 3, 131–137.

Tamminen, M. 1985. Two levels are as good as any. *Journal of Algorithms 6*, 1, 138–144.

The Unicode Consortium. 1991. *The Unicode Standard: Worldwide Character Encoding. Version 1.0*, Volume 1 and 2. Addison-Wesley.