When can we sort in $o(n \log n)$ time?

Amir M. Ben-Amram* and Zvi Galil**
Tel-Aviv University — Tel-Aviv University and Columbia University

Socrates: What sort? Not a trivial one, as I see it
Plato (H. Tredennick, tr.)

SUMMARY OF RESULTS

We define two conditions on a random access machine (RAM) with arithmetic and Boolean instructions and possible bounds on word and memory sizes. One condition asserts that we either restrict attention to short words or allow non-uniform programs. The second asserts that we either allow a large memory or a double-precision multiplication.

Our main theorem shows that the RAM can sort in $o(n \log n)$ time if and only if both of these conditions hold. This theorem breaks down into four upper bounds only one of which has been known before, and two lower bounds neither of which has been known.

1. Introduction and Overview

The problem of sorting is one of the most fundamental, and well studied, problems in computer science. For the most general formulation of the problem it is natural to consider the comparison tree model, in which it is well known that $\Theta(n \log n)$ comparisons are necessary and sufficient to sort an arbitrary n-tuple [1,10]. On the other hand, the random access machine model of computation, which is considered the most realistic model of conventional computers, is very different. The data handled are integer numbers, and a set of specialized instructions is available to handle them. In addition to comparison, we assume arithmetic/logical capabilities and the feature of indirect addressing. These features are known to allow sorting algorithms that transcend the decision tree model, and may achieve sub-$n \log n$ performance. We call a sorting algorithm *fast* if it can sort any array of n integers in $o(n \log n)$ time. The existence of fast sorting algorithms is known for a long time, the simplest example being *bucket sort* [10]. This algorithm sorts n integers in the range $[0, n)$ in $O(n)$ time. We are interested in algorithms that can handle every representable integer; therefore the simple bucket sort will be useful only if the words (memory registers) of our RAM are approximately $\log n$ bits long. Such RAM provides an

* Work supported by a Charles Clore fellowship.

** Work partially supported by NSF Grant CCR-90-14605 and CISE Institutional Infrastructure Grant CDA-90-24735.

example of a machine model that allows fast sorting. In contrast, Paul and Simon proved in [12] that $\Omega(n \log n)$ time is required for sorting n integers on a RAM with unbounded words and an arithmetic instruction set (addition, subtraction and multiplication). This result showed that fast sorting is not always possible, even on a RAM.

In this paper, we study machines of a finite word-length. We denote the word size by w. For giving the bounds on word length, we use the "dotted O" notation introduced by Ben-Amram [4]. $f = \dot{\Omega}(g)$ if there exists a constant c such that (when $n \to \infty$) $\liminf \frac{f(cn)}{g(n)} > 0$. $f = \dot{o}(g)$ if $\limsup \frac{f(n)}{g(cn)} = 0$ for all c. Note that for functions of polynomial growth, i.e., functions f such that $n^a \leq f \leq n^b$ for some $0 < a < b$, the dotted Ω and o coincide with their undotted counterparts.

Careful analysis of Paul and Simon's proof shows that it extends to finite words given $w = \dot{\Omega}(2^{n \log n})$.

Definition. *A RAM is said to have the short word property if* $w = \dot{o}(2^{n \log n})$.

The *basic instruction set* for the RAMs of this paper contains both arithmetic and Boolean instructions. The arithmetic instructions include addition, subtraction and multiplication, which are all performed modulo 2^w. Boolean operations are bitwise AND, NOT etc. Note that the Paul-Simon bound does not apply to our model, because of the Boolean instructions. Apart from the above we have the usual instructions for comparison, assignment using direct and indirect addresses etc.

Another characteristic of the model is the address size a. The RAM's memory is composed of 2^a words, addressed by integers in the range $[0, 2^a)$. We assume that $a \leq w$ (a memory address fits within a word), and $a > \log n$ (so internal sort is possible). We say a program runs in linear space if the memory size it requires is linear in n; i.e., $a = \log n + O(1)$. Any given linear-space algorithm will be considered usable: that is to say, we are not interested in constant-factor reductions in space (which are possible, by the way, for all sorting algorithms). Note that the choice of memory size as a measure of space avoids the usual abuse of the unit-cost criterion which counts cells accessed, no matter what range of addresses is required.

Definition. *A RAM is said to have the large memory property if* $w - a = o(\log n)$.

The field of sorting algorithms was revolutionized in recent years with the discovery of Fusion Tree sorting by Fredman and Willard [8]. These authors paid careful attention to the requirements they place on the machine, trying to minimize demands, and thus illuminated the challenge: find the exact properties that are necessary and sufficient for fast sorting. The Fusion Tree algorithm has two main advantages. First, this algorithm is fast for every word length. Second, it runs in linear space. To achieve this performance, the instruction set was slightly extended beyond our basic set. The extension they chose is a "double precision feature," which allows multiplication to produce a two-word product. This choice is intended to be as conservative as possible: most actual computers support this operation, and it may replace some uses of more general instructions (like shift). For example, upon multiplying x by 2^{w-r}, we obtain the result of right-shifting x by r bits (in the high-order word produced). Note that double-precision addition and subtraction need not be considered as special extensions, because in these operations, there is only one bit of carry into the high-order word, and the whole operation can be implemented using single-precision addition and subtraction, plus a few tests to determine the value of the carry.

They also consider the question of uniformity (freedom of the program from dependence on n and w). Their algorithm is not uniform: some "magic numbers" dependent on w have to be provided. We call this feature *weak non-uniformity*, when these numbers are hard-coded in the program, which is otherwise fixed. A *strongly non-uniform* program makes use of preset tables which may be as large as memory allows, and then the program code may also depend on problem parameters; this is not an essential extension since we could run a program stored in memory. It is certainly better for an algorithm, if non-uniform, to be so weakly. Note that a uniform program may compute any function of n and w, so non-uniformity is only essential if we cannot compute the necessary constants or tables fast enough.

In these paper we tackle the challenge of finding the properties necessary and sufficient for fast sorting. For easy reference, we label the properties considered as follows:

$\mathcal{NU}$ a non-uniform program.

$\mathcal{SW}$ a short word.

$\mathcal{DP}$ a double-precision multiply instruction.

$\mathcal{LM}$ a large memory.

The results of our research suggest that these four properties combine to determine the possibility of fast sorting in a very simple way.

Conjecture. *A RAM can sort in $o(n \log n)$ time if and only if it satisfies $(\mathcal{NU} \vee \mathcal{SW}) \wedge (\mathcal{DP} \vee \mathcal{LM})$.*

Proving this statement breaks down to showing the sufficiency of four pairs of properties, and the insufficiency of two. The first of the four positive claims follows from [8]; we prove the rest in this paper. Among the negative statements, one is proved completely, and the other is proved under a certain restriction, that we discuss below.

Theorem 1 ($\mathcal{NU} \wedge \mathcal{DP}$). *A RAM with double-precision multiply can sort n integers in $o(n \log n)$ time, using weak non-uniformity and linear space.*

This theorem is proved in [8].

Theorem 2 ($\mathcal{SW} \wedge \mathcal{DP}$). *A RAM with double-precision multiply and a short word can sort n integers in $o(n \log n)$ time with a uniform program and linear space.*

To prove this theorem, we show (in Section 2) how to construct in $o(n \log n)$ time the non-uniform constants needed by the Fusion Tree algorithm: with this addition, the latter becomes a uniform program and satisfied the theorem.

Theorem 3 ($\mathcal{NU} \wedge \mathcal{LM}$). *A RAM of any word length can sort n integers in $o(n \log n)$ time, using strong non-uniformity and a large memory.*

The proof of this theorem falls in two parts: (1) the case $a = w$ (where we have largest possible memory) is handled by implementing the Fusion Tree algorithm such that double-precision multiplication is replaced by lookup in a very large table. (2) When a is smaller, a combination of radix sort and the solution for (1) is used. We prove this theorem in Section 3.

Theorem 4 ($\mathcal{SW} \wedge \mathcal{LM}$). *A RAM of short word and large memory can sort n integers in $o(n \log n)$ time with a uniform program.*

This theorem is proved by combination of two algorithms. In Section 7 we describe an implementation of Kirkpatrick-Reisch sort [9] which satisfies the theorem with a certain restriction on the word length. Larger words are handled by a modified Fusion Tree (Section 8). The modifications remove most of the uses of double-precision multiplication, leaving only its application to moving a small group of bits from the left end of one word to the right end of another. When this group is small enough (less than $\log n$ bits), the operation can be replaced by table lookup, and the table will be small enough to be built by a uniform algorithm.

Theorem 5 ($\neg\mathcal{DP} \wedge \neg\mathcal{LM}$). *A RAM of the basic instruction set and an address length $a = w - \Omega(\log n)$ requires $\Omega(n \log n)$ time to sort n numbers for all word lengths, even if strong non-uniformity is allowed.*

The proof is given in Section 4. It uses a general lemma, which models a RAM program as a decision tree, and shows that for appropriate sets of problem instances, different answers must be obtained at different leaves. The time bound then follows in the usual way.

Theorem 6 ($\neg\mathcal{NU} \wedge \neg\mathcal{SW}$). *A uniform RAM program requires $\Omega(n \log n)$ time to sort n numbers given $w = \dot{\Omega}(2^{n \log n})$, and $a \leq w/2$, even if a double-precision multiply instruction is included.*

Note the restriction on memory size which should be removed for completing the proof of our conjecture. We remark that this restriction lies between the large memory requirement of Theorems 3 and 4 and the linear requirements of the first two upper bounds.

The proof of the last theorem uses a lemma, proved in Section 5, which extends the methods of the former section to cover possible usage of double-precision (and some other) operations.

Section 6 presents an application of the same lemma to a problem other than sorting (computing the function $\log x$).

Note that the order of proofs does not match the order of theorems in this section; the proof of Theorem 4 was deferred since the Modified Fusion Tree algorithm is the longest and most complicated part of this paper.

We number the bits of a word from least significant (bit 0) to most significant (bit $w - 1$). The following fact is frequently used in our algorithms.

Fact 1. *Using arithmetics module 2^w, the expression -2^t yields a mask that selects bits $t, t + 1, \ldots, w - 1$.*

2. Proof of Theorem 2

This theorem considers a RAM equipped with the double-precision multiply instruction, whose word length satisfies $w = \dot{o}(2^{n \log n})$. A uniform sorting algorithm for this model is obtained from the Fusion Tree algorithm by removing its use of non-uniform constants. To this end, we consider one by one the constants appearing in the algorithm (recall that only a fixed number of such constants is used). We show how to compute each of them by a uniform program, given n and w as input, within $o(n \log n)$ time. The techniques shown here will also serve for a similar purpose in later sections. The fast sorting algorithm is obtained as follows: first, the procedures described in this section are executed. They compute required constants in $O(\log w) = o(n \log n)$ time. Then, the Fusion Tree algorithm is executed without change. The whole program is uniform, because the constants needed for the second part are computed in the first.

It is necessary to consult [8] for verifying the completeness of this proof (namely, that the constants that we claim to suffice for implementing Fusion Trees do so indeed). We do not attempt to make our proof self-contained, since this would necessitate repeating the contents of [8]. We remark that the following sections are self-contained.

The key constant in the algorithm is B, the approximate degree of an internal node. To guarantee the desired running time, this constant should be bounded by (and sufficiently close to) $\min(w^{1/6}, 2^{\sqrt{\log n}})$. The procedure to find B is given in Appendix A. The procedure will give B a value which obeys the above bound and is not less than its half. Its running time is proportional to $\log B \le \sqrt{\log n}$.

The algorithm performs operations on words divided into B fields which have to include approximately w/B bits each. We find the size of each field, called F, by repeated doubling (as in the procedure for B); this takes $O(\log F) = O(\log w)$ time, and is accurate up to a factor of two, which is enough.

For operations on these aggregated words, the algorithm uses two special constants. The first is 2^{F-1}, which is obtained in $O(\log F)$ time by a series of squarings and multiplications. The second is a word G containing 2^{F-1} in each of its B fields (a single one-bit at the left end of each field). The procedure to find G appears in Appendix A. This value is obtained in $O(\log B)$ operations.

The most significant trick in the Fusion Tree algorithm is the computation of MSB (most significant bit) position and masks. This procedure uses several interesting constants. We choose an even number s such that $\sqrt{w} + 1 \le s \le 2\sqrt{w}$. The procedure uses aggregates which are words divided into fields of s bits each. (Here we change a little from the description in [8], allowing more freedom in choosing s, and refraining from double-precision constants. The required change is that the value of the most significant block will be cut in two halves of $s/2$ bits before the table lookup that finds its log. The constants listed below match our modifications). The number of fields is $t = \lceil \frac{w}{s} \rceil$, where one field (either the rightmost or the leftmost) may be shorter: its length is $r = w - s(t-1)$. Choosing s by repeated doubling takes $O(\log w)$ time, yields an appropriate result and also facilitates the construction of the other constants we need using procedures of the kind used for B and G. The constants are:

$$S_1 = 2^{s-1}, \quad S_2 = 2^{s/2-1}, \quad S_3 = 2^{r-1}, \quad S_4 = 2^{t-1},$$
$$S_5 = 2^{w-s}, \quad S_6 = 2^{w-s/2}, \quad S_7 = 2^{w-r}$$

(and combinations thereof);

$$C_1 = \underbrace{10\cdots0}_{r}\ \underbrace{100\cdots0}_{s}\ \cdots\ \underbrace{100\cdots0}_{s}\ \underbrace{100\cdots0}_{s}$$
$$C_2 = \underbrace{100\cdots0}_{s-1}\ \underbrace{100\cdots0}_{s-1}\ \cdots\ \underbrace{100\cdots0}_{s-1}\ \underbrace{100\cdots0}_{r-1}$$
$$C_3 = \underbrace{0}_{r}\ \underbrace{2^{t-1}}_{s}\ \cdots\ \underbrace{2}_{s}\ \underbrace{1}_{s}$$
$$C_4 = \underbrace{1}_{s}\ \underbrace{2}_{s}\ \cdots\ \underbrace{2^{t-1}}_{s}\ \underbrace{0}_{r}$$

C_2 is used to "perfectly compress" the lead bits of the fields (the bits that are on in C_1). C_3 and C_4 are used as "pocket size" tables. Their structure is similar to that of G (and hence they are similarly built), for example C_3 equals the following right-justified value:

$$\underbrace{100\cdots0}_{s+1}\ \underbrace{100\cdots0}_{s+1}\ \cdots\ \underbrace{100\cdots0}_{s+1}\ 1 \quad .$$

3. Proof of Theorem 3

Here the use of double-precision multiplication in Fusion Trees is replaced with the help of a large table, which has to be preset in memory. Thus strong non-uniformity and a large memory are required.

As an intermediate step, we replace the double-precision multiply instruction with a new instruction, the *swap* instruction. This instruction swaps the two halves of a word (assume for simplicity w is even). We show below that with *swap*, we can implement double-precision multiplication in constant time. The importance of this transition for our purposes is that *swap* is a monadic operator. Thus, it can be simulated by a table SWAP indexed by the operand, such that $\mathrm{SWAP}(x)$ is the result of swapping the half-words of x. The table's size is 2^w words, which is a bit too much, so we reduce it to $2^{w/2}$ in the following way. The table will record $\mathrm{SWAP}(x)$ only for x which is a multiple of $2^{w/2}$. For general x, the swap instruction will be simulated by the expression:

$$2^{w/2}x + \mathrm{SWAP}(x \wedge (-2^{w/2}))$$

whose correctness is easily verified.

The following lines carry out the double-precision operation

$$(H, L) \leftarrow A \times B$$

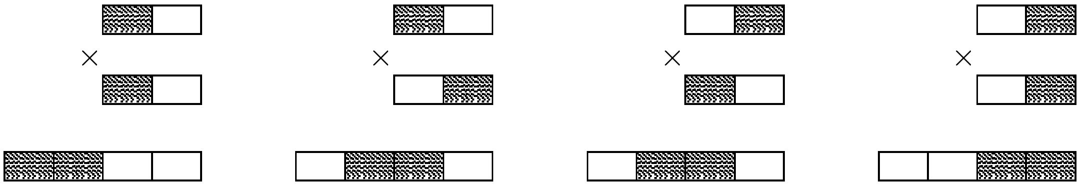


Figure 3.1

(yielding a high-order word H and a low-order word L) using the *swap* operation (represented by the operator symbol $\sim$). Figure 3.1 illustrates the computation. In the sixth line, double-precision addition is used so that H receives the carry from the addition that produces L. Double-precision addition can be simulated with ordinary addition and a few tests; we omit the details.

$$
\begin{aligned}
M &\leftarrow 2^{w/2} - 1 \\
T_1 &\leftarrow (A \wedge M) \times (B \wedge M) \\
T_2 &\leftarrow (A \wedge M) \times (\sim B \wedge M) \\
T_3 &\leftarrow (\sim A \wedge M) \times (B \wedge M) \\
T_4 &\leftarrow (\sim A \wedge M) \times (\sim B \wedge M) \\
(H, L) &\leftarrow T_1 + \sim(T_2 \wedge M) + \sim(T_3 \wedge M) \\
H &\leftarrow H + (\sim T_2 \wedge M) + (\sim T_3 \wedge M) + T_4
\end{aligned}
$$

This solution is only valid if $a = w$, because the table is indexed by numbers of w bits. Hence if $a = w$ we are done: the swap operation is simulated by the table, double-precision multiplication is simulated by *swap* and the Fusion Tree algorithm can be used. If $a < w$, we implement the swap operation for operands of a bits (assume a is even here; otherwise we use $a - 1$). This allows us to run the Fusion Tree algorithm, without further modification, on the "slice" consisting of the rightmost a bits of each number. The standard radix-sort technique shows that after we sort the data by this value, we can continue by splitting the sorted list according to bit $a + 1$ (and arranging those with a zero bit before those with a one), then according to bit $a + 2$ etc; after handling bit w, the list will be sorted. Since $a = w - o(\log n)$, there will be $o(\log n)$ such rounds. Testing the kth bit of a word is accomplished in constant time by ANDing it with 2^{k-1}, and it clearly follows that each round takes $O(n)$ time for a total of $o(n \log n)$ time in the radix-sort rounds. Since the Fusion-Tree sort takes $o(n \log n)$ time the proof is complete.

4. Lower Bounds for Non-Uniform programs

This section gives a lower-bound technique, applies it to prove Theorem 5, then extends it to a stronger model than the one considered in the theorem.

4.1 Proof of Theorem 5

Theorem 5 states that with the basic instruction set, large memory is essential for fast sorting, even for strongly non-uniform programs. We thus assume that the address size a satisfies $w - a = \Omega(\log n)$, and the 2^a memory words are all preset to values that may depend on n and w, as may the program itself.

A central idea in the lower bound proof is dividing each word in two parts. One consists of the rightmost a bits, and is called *the internal part*. The rest of the word is *the external part*. We denote the internal part of x by $R(x)$ and the external part by $L(x)$. We denote the memory word whose address is m by $\langle m \rangle$.

A program in our model can be represented as a tree $\mathcal{T}$ in which each node is labeled with an instruction of the program. The root of the tree represents the first instruction of the program, ignoring input operations: we assume the input to be present in a certain array. We have five types of nodes:

1) An assignment node. Such node has one child and is labeled with the instruction $\langle i \rangle \leftarrow j$ or $\langle i \rangle \leftarrow \langle j \rangle$ where i, j are constants.
2) A computation node. Such node has one child and is labeled with $\langle i \rangle \leftarrow \langle j \rangle \circ \langle k \rangle$, where i, j, k are constants and $\circ \in \{+, -, \times, \wedge, \vee, \neg\}$ ($\neg$ ignores the first operand). We do not allow a computation to involve a constant, but the memory cells used might be set to constant values with an assignment first.
3) An indirect addressing node. This node is labeled with the operation $\langle i \rangle \leftarrow \langle\langle j \rangle\rangle$ or $\langle\langle j \rangle\rangle \leftarrow \langle i \rangle$, where i, j are constants.
4) A decision node: this node has two children and is labeled with a condition that selects one of them for the next step. A typical condition is "$\langle i \rangle > \langle j \rangle$?"
5) A halt node: this is the type of leaves. When a leaf is reached, the input array should be sorted.

The *internal configuration* of the program at a certain node consists of the internal part of all memory words when this node is reached.

The Tree Invariant. *The internal configuration at each node of $\mathcal{T}$ determines the one at its children.*

Proof. The argument depends on the type of operation represented by the node. We abbreviate "internal configuration" to IC.

(i) An assignment node. The IC at the successor of this node differs from the current only on the internal part of $\langle i \rangle$. For the first form, this part is changed into a constant; for the second, it is changed into $R(\langle j \rangle)$. Since j is a constant, $R(\langle j \rangle)$ depends only on the current IC.

(ii) A computation node. The IC at the successor of this node differs from the current only on the internal part of $\langle i \rangle$. It is easy to verify that for each of the operations considered, $R(x \circ y) = R(R(x) \circ R(y))$. As a result, the internal part of $\langle i \rangle$ is determined by $R(\langle j \rangle)$ and $R(\langle k \rangle)$, depending only on the current IC.

(iii) An indirect addressing node of the first kind (load). The configuration at the successor node differs from the current one only on $R(\langle i\rangle)$, which is set to to $R(\langle\langle j\rangle\rangle)$. The current IC determines both the address of the cell accessed (as an address is restricted to a bits, it equals $R(\langle j\rangle)$) and the new contents of $R(\langle i\rangle)$ (the current internal part at that address).

(iv) An indirect addressing node of the second kind (store). The configuration at the successor node differs from the current one only on $R(\langle\langle j\rangle\rangle)$. The identity of this cell is determined by the current IC, because the address is $R(\langle j\rangle)$; the new contents of the internal part is $R(\langle i\rangle)$, which depends only on the current IC.

(v) Decision nodes and halt nodes do not change the memory contents. □

We call a set of problem instances (input vectors) *external* if all of them agree on their internal part. We call the set *distinguishable* if the program must end in a different internal configuration for each instance.

Lemma 1. *Let S be an external and distinguishable set of problem instances. The worst-case running time of a program to solve the problem on the model considered in Theorem 5 is at least* $\log|S|$.

Proof. We use the tree representation of the program. As the computation starts, the contents of memory are fixed except for the input array. Since S is external, the internal portion of these words is the same for every instance. Thus the root of the tree is associated with a unique internal configuration. By the tree invariant it follows that the internal configuration at every node, and in particular at every leaf, is also independent of the particular instance. Since S is distinguishable, the program must reach a different leaf on each instance in S. Thus there are at least $|S|$ leaves, and as the tree is binary, its height is at least $\log|S|$. □

This is a general result, applicable to any problem for which such a set of instances may be built. The following lemma implies the lower bound for sorting.

Lemma 2. *Let c be a constant such that $w - a \geq c\log n$. Let $d = 1$ if $c \geq 1$, and $c/2$ otherwise. There exists a distinguishable and external set of problem instances S for sorting such that* $\log|S| = (1 - o(1))d\,n\log n$.

Proof. We construct vectors $\mathbf{x}$ that will serve as problem instances. We begin with the case $c \geq 1$, where the external part of each memory word contains at least $\log n$ bits. Recall that $a \geq \log n$, so the internal part contains at least $\log n$ bits as well. Let S_n be the set of permutations on $\{0, 1, \ldots, n-1\}$. For $\pi \in S_n$ we define $\mathbf{x}_\pi$ as follows:

$$\mathbf{x}_\pi(i) = \begin{array}{|c|c|c|c|} \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{a} & \multicolumn{1}{c}{0} \\ \hline 0 & \pi(i) & 0 & i \\ \hline \end{array} \qquad \text{for } i = 0, 1, \ldots, n-1$$

Let $S = \{\mathbf{x}_\pi : \pi \in S_n\}$. The set S is both external (all vectors have the same internal part) and distinguishable (after sorting, the internal part will equal π^{-1}). The lemma follows since $\log|S| = \log n! = (1 - o(1))n\log n$.

We now consider the case $c < 1$. Here only n^c different values can appear in the external part of a word (assume for simplicity that n^c is integral). Consider a vector $\mathbf{x}$ whose external part contain values from $[0, n^c - 1)$, while the internal part equals $(0, 1, \ldots, n-1)$. The elements of $\mathbf{x}$ are all different, and can be recognized by looking at their internal parts. Let $1 \le i < j \le n$. If x_i comes after x_j in sorted order, necessarily $L(x_i) > L(x_j)$. Otherwise $L(x_i) \le L(x_j)$. Now assume that $n^c \le {}^n\!/_2$ (this is true for n large enough). Let $\pi : [0, {}^n\!/_2) \to [0, n^c)$ be surjective; there are over $(n^c)^{\frac{n}{2} - n^c}$ such functions. Define $\mathbf{x}_\pi$ by

$$\mathbf{x}_\pi(i) = \begin{cases} \begin{array}{rcc} a & & 0 \\ \hline \multicolumn{1}{|r|}{\pi(i)} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{i} \\ \hline \end{array} & \text{for } i = 0, 1, \ldots, {}^n\!/_2 - 1 \\ \begin{array}{rcc} a & & 0 \\ \hline \multicolumn{1}{|r|}{\pi(n-1-i)} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{i} \\ \hline \end{array} & \text{for } i = {}^n\!/_2, \ldots, n-2, n-1 \end{cases}$$

The above arguments show that the sorted order of $\mathbf{x}$ determines π. We thus obtain an external and distinguishable set S whose size is over $(n^c)^{\frac{n}{2} - n^c}$. Hence $\log|S| \ge (1 - o(1))\frac{c}{2} n \log n$. $\square$

4.2 An Extended Model

We strengthen our lower bound by extending the instruction set of the RAM considered with shift instructions and arbitrary conditionals. Formally, we define RAM1 to be a RAM with word length w, address length a, and an instruction set that includes:

1. addition, subtraction and multiplication modulo 2^w.
2. all bitwise Boolean instructions.
3. the following shift instructions:

   arithmetic left shift moves the contents of the word a single bit position to the left, putting 0 in the rightmost position.

   arithmetic right shift moves the contents of the word a single bit position to the right, putting 0 in the leftmost position.

   circular left shift transfers the leftmost bit of the input word into the rightmost bit of the result and moves the rest of the word a single bit position to the left.

   circular right shift transfers the rightmost bit of the input word into the leftmost bit of the result and moves the rest of the word a single bit position to the right.
4. an arbitrary set of conditionals.

Denote by $[x]_i$ the value of bit i of the word x $(i = 0, \ldots, w-1)$.

Lemma 3. *Let S be an external and distinguishable set of problem instances. The worst-case running time of a (strongly non-uniform) RAM1 program that solves this problem is at least* $\log|S|$.

Proof. We use the tree representation of the program, as above, modified according to the following rules:

(i) each computation node that performs a right-shift on $< k >$ is preceded by a decision node which tests the value of $[< k >]_a$ (we make use of the allowance for arbitrary conditionals!).

(ii) each computation node that performs a circular left-shift on $< k >$ is preceded by a decision node which tests the value of $[< k >]_{w-1}$.

These decision nodes cause the program to branch according to the only bit outside the internal part which is moved into the internal part by the shift instruction.

We claim that the *tree invariant* holds for the resulting tree. In fact, the proof of the tree invariant has only to be extended to cover the additional operations of RAM1. To begin, the allowance for arbitrary conditionals is harmless since decision nodes do not affect memory contents anyway.

Consider now a right-shift operation on $< k >$. The internal part of the result consists of bits 1 through a of the operand. The value of bit a is fixed by the branch taken in the preceding decision node (rule (i)). Bits 1 through $a-1$ are determined by $R(< k >)$. The tree invariant is maintained. Similar arguments hold for the other shift instructions.

The lemma follows from the tree invariant as before. Note that we added new decision nodes when building the tree (rules (i) and (ii)), but each such node precedes a shift instruction; and since the lower bound of $\log |S|$ only accounts for decision nodes, it applies to the running time of the original program (where both decisions and shifts are counted).
□

Corollary 1. *If $w - a = \Omega(\log n)$, a (strongly non-uniform) RAM1 program for sorting takes $\Omega(n \log n)$ time.*

Proof. Combine Lemma 3 with Lemma 2. □

5. Lower Bounds for Uniform Programs

In this section, we extend the techniques of the last section to a model that includes double-precision operations, at the price of requiring uniformity, and a certain restriction on the form of the input sets. A lower bound for sorting follows in a similar way, proving Theorem 6.

We consider the model RAM2, which adds to RAM1 the capability of double-precision addition, subtraction and multiplication. We only consider uniform programs.

5.1 A General Result

The uniformity assumption implies that the program may only contain a finite number of constants. We denote by $\mathcal{C}$ the set of constants. For an integer x, let $\ell(x) = \lceil \log(x+1) \rceil$ (length of the binary representation of x). Define $L_0 = \max\{\ell(c) : c \in \mathcal{C}\}$.

In essence, this section repeats the arguments of the former, but more effort is required since the tree invariant defined there is defeated by double-precision multiplication. In order to handle this operation, we ensure that all numbers handled by the program have a special form. This form is defined by the following figure (which represents a word):

q	p	l	0
$\sigma\sigma\sigma\sigma\sigma$	///	$\tau\tau\tau\tau\tau$	///

σ and τ stand for bit values of either 0 or 1 (so bits l through $p-1$ are all equal, and so are bits q through $w-1$). The dashed areas represent bit strings of arbitrary value. The values of l, p and q will not be fixed, but vary from step to step (we give the exact values later). We denote the set of numbers having the above form (called *nice* numbers) by $N(l,p,q)$. If both σ and τ are zero, the numbers are *nicer*. The set of nicer numbers is denoted by $N^*(l,p,q)$.

We impose the following condition on the parameters used for defining nice sets:

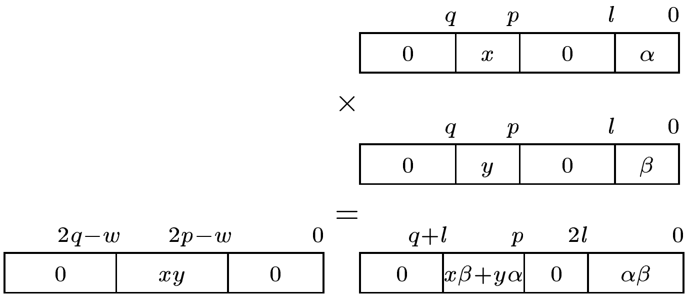
$$0 < l < p < q < w; \quad 2l \leq p; \quad 2p \geq w + a; \quad q + l \leq w\,. \tag{$*$}$$

While unrolling the program into a computation tree, we perform a few transformations: first, for shift instructions, we apply rules (i) and (ii) of the last section (preceding them with tests). Next, consider double-precision addition and subtraction: note that in these operations, a single carry bit propagates into the high-order word. We precede each of these instructions with a decision node that branches according to the value of this carry bit. In each branch, we may replace the double-precision operation by a single-precision one, followed by an assignment of a constant to the high-order word of the result.

Finally, we replace each multiplication operation by the procedure MUL, given in Appendix B. This procedure reduces double-precision multiplication of nice numbers to (double-precision) multiplications of nicer ones. (For simplicity we assume that all multiplications in the original program are done in double precision).

Lemma 4. *All instructions of RAM2 satisfy the following two claims. (i) if the operands belong to $N(l,p,q)$ (for multiplication, to $N^*(l,p,q)$), then the word (or words) of the result belongs to $N(2l, 2p-w, q+l)$. (ii) under the same conditions, the internal part of the result word(s) is determined by the internal parts of the operands.*

Proof. Consider (double-precision) multiplication first. We make use of the relations shown in the following diagram:



Part (i) of the lemma is immediate. To verify (ii), Recall that $2p \geq w + a$, so both p and $2p-w$ are at least a. If $l \leq a$, the internal part of the low-order word of the product equals $\alpha\beta$, and that of the high-order word is zero. Thus, the internal part of both words of the result is determined by the internal parts of the multiplicands. If $l > a$, the property holds since $R(\alpha\beta) = R(R(\alpha)R(\beta))$.

For the other instructions (addition, subtraction, Boolean operations and shifts), the required properties are immediate and may be verified by the reader. $\square$

In what follows we refer to a constant ε such that $3/4 < \varepsilon < 1$. Let $\gamma = 2\varepsilon - 1$ and $\delta = 2\gamma - 1$. Note that $0 < \delta < \gamma < \varepsilon$. All w larger than some constant (dependent on ε) satisfy the relations $w \geq 2^{1/(1-\varepsilon)}$, $w > (\log w)^{1/\delta} + 1$ and $w > (L_0)^{\delta}$. We assume these relations to hold (a valid assumption, since we are not interested in values of w bounded by a constant). Note that the first relation implies $w^{\varepsilon-1} \leq 1/2$.

For $k \geq 0$, we define

$$\begin{aligned} p_k &= w - 2^{k-1}\lfloor w^{\gamma} \rfloor \\ l_k &= 2^k \lfloor w^{\delta} \rfloor \\ q_k &= p_0 + l_k \end{aligned}$$

Lemma 5. *Let $t \leq (1-\varepsilon)\log w$. Assume that $w > (\log n)^{1/\delta} + 1$ and $a \leq w - w^{\varepsilon}$. For $0 \leq k < t$, the parameters l_k, p_k and q_k satisfy $(*)$. Further, if the input to the program is from $N(l_0, p_0, q_0)$, then after $k < t$ program steps all numbers in memory are in $N(l_k, p_k, q_k)$.*

Proof. It is easy to see that l_k and q_k increase with k, while p_k decreases. Therefore, to verify $(*)$ for all $0 \leq k < t$, it suffices to verify the following:

$$\begin{aligned} &0 < l_0, \quad p_0 < q_0, \\ &2l_{t-1} \leq p_{t-1}, \quad 2p_{t-1} \geq w + a, \quad q_{t-1} + l_{t-1} \leq w\,. \end{aligned}$$

The first two relations are immediate. Let us verify the last three:

$$\begin{aligned} 2l_{t-1} - p_{t-1} &= 2^t \lfloor w^{\delta} \rfloor - w + 2^{t-2}\lfloor w^{\gamma} \rfloor < \frac{5}{4}\lfloor w^{\delta} \rfloor - w \\ &\leq \frac{5}{4} w^{1-\varepsilon+\delta} - w = \frac{5}{4} w (w^{3\varepsilon-3} - \frac{4}{5}) \leq \frac{5}{4} w (\frac{1}{8} - \frac{4}{5}) < 0 \end{aligned}$$

so $2l_{t-1} \leq p_{t-1}$. Next,

$$2p_{t-1} = 2w - 2^t \lfloor w^{\gamma} \rfloor \leq 2w - w^{1-\varepsilon+\gamma} = 2w - w^{\varepsilon} \geq w + a$$

and finally

$$\begin{aligned} q_{t-1} + l_{t-1} &= p_0 + 2l_{t-1} = w - \frac{1}{2}\lfloor w^{\gamma} \rfloor + 2^t \lfloor w^{\delta} \rfloor \\ &\leq w - \frac{1}{2}\lfloor w^{\gamma} \rfloor + w^{1-\varepsilon+\delta} < w + 1 - \frac{1}{2} w^{\gamma} + w^{1-\varepsilon+\delta} \leq w + 1 - \frac{1}{2} w^{\gamma} + w^{2\gamma-\varepsilon} \\ &= w + 1 - w^{\gamma}(\frac{1}{2} - w^{\varepsilon-1}) \leq w + 1 \end{aligned}$$

so $q_{t-1} + l_{t-1} \leq w$.

The second claim is proved by induction. When the program starts, only the values of n, w and the input vector reside in memory. The input vector is assumed to satisfy the claim. The value of w take up $1 + \lfloor \log w \rfloor$ bits; the assumption $w > (\log w)^{1/\delta} + 1$ guarantees that that this number is bounded by l_0. Similarly, the assumption $w > (\log n)^{1/\delta} + 1$ guarantees that the value of n fits within the rightmost l_0 bits. Thus, $n, w \in N(l_0, p_0, q_0)$.

Assume the claim to hold after $k-1$ steps. It is easy to verify that

$$N(l_{k-1}, p_{k-1}, q_{k-1}) \subseteq N(l_k, p_k, q_k)$$

Thus all numbers which are not affected in step k continue to satisfy the claim. The only instructions that enter a new value in memory are an assignment instruction involving a constant, and a computation instruction. Regarding the latter, Lemma 4 justifies the claim since

$$(l_k,\ p_k,\ q_k) = (2l_{k-1},\ 2p_{k-1} - w,\ q_{k-1} + l_{k-1}) \ .$$

As for the former, all constants of the original program are of L_0 bits at most, and belong to $N(l_0, p_0, q_0)$. The MUL procedure includes new constants, which depend on the parameters l, p, q: for example, 2^q. The form of these constants depends on the parameters. It is necessary to verify specifically that each of them belongs to $N(l_k, p_k, q_k)$, where k is the step number in which the procedure is entered (this means that the parameters are set to l_{k-1}, p_{k-1} and q_{k-1} respectively). In the above example, we obtain $2^{q_{k-1}}$. This is a word with a single one-bit, in position q_{k-1}. This word belongs to $N(l_k, p_k, q_k)$ because the one-bit falls into the range $[p_k, q_k)$. $\square$

Lemma 6. *Let S be an external and distinguishable set of problem instances, which consists solely of values from $N(l_0, p_0, q_0)$. Let $t = (1-\varepsilon)\log w$, and assume $w > (\log n)^{1/\delta} + 1$ and $a \le w - w^{\varepsilon}$. The worst-case running time of a uniform RAM2 program that solves the problem is $\Omega(\min(t, \log |S|))$.*

Proof. Unroll the program into a tree, while making the modifications described above; call the resulting computation tree $\mathcal{T}$. It describes a program which is slower than the original one by at most a constant factor (the running time of MUL). Assume that the running time of the resulting program is bounded by t. The last two lemmas combine to show that the tree invariant of the last section is maintained in $\mathcal{T}$. Therefore, as S is external and distinguishable, the number of leaves in $\mathcal{T}$ must be at least $|S|$, and its height at least $\log |S|$. $\square$

5.2 A Lower Bound for Sorting

The next theorem gives a lower bound for sorting on RAM2. Theorem 6 is a corollary.

Theorem 7. *Assume that $a \le w - w^{\varepsilon}$, for some $3/4 < \varepsilon < 1$. A uniform RAM2 program requires $\Omega(\min(\log w, n \log n))$ time to sort.*

Proof. Let $t = (1-\varepsilon)\log w$, δ and γ as above. Since the statement is trivial when $\log w \le n$, we assume $w > 2^n$. Therefore, we for n large enough, $w > (\log n)^{1/\delta} + 1$.

Consider the set $S = \{\mathbf{z}_\pi : \pi \in S_n\}$ where

$$\mathbf{z}_\pi(i) = \begin{array}{cccc} & \qquad\ \ p_0 & & \qquad 0 \\ \hline \multicolumn{1}{|c|}{\quad 0 \quad} & \multicolumn{1}{c|}{\pi(i)} & \multicolumn{1}{c|}{\quad 0 \quad} & \multicolumn{1}{c|}{\ i\ } \\ \hline \end{array} \qquad (i = 0, 1, \ldots, n-1)$$

$p_0 = w - {}^1\!/_2\lfloor w^\gamma \rfloor > a$, hence the set S is both external and distinguishable. $|S| = n!$ so $\log|S| = \Omega(n \log n)$. The relation $w > (\log n)^{1/\delta} + 1$ implies that $q_0 - p_0 = l_0 = \lfloor w^\delta \rfloor \geq \lfloor \log n \rfloor + 1$, the number of bits in $\pi(i)$, which shows that $\mathbf{z}_\pi(i) \in N(l_0, p_0, q_0)$. It now follows from Lemma 6 that a program that sorts all the instances in S takes $\Omega(\min(t, \log|S|)) = \Omega(\min(\log w, n \log n))$ time. $\square$

6. Lower Bounds for Function Computations

In this section we demonstrate that the techniques of the last section have some applications besides sorting. We consider the problems of computing the function $\lceil \log x \rceil$ and the function $\lceil \log\log x \rceil$ for an arbitrary x. We give tight time bounds of $\Theta(\log w)$ and $\Theta(\log\log w)$, respectively, for computing these functions on RAM2 (with certain memory constraints).

These functions were considered by Bshouty [6] in a very similar model. Bshouty gave weaker time bounds for these problems, namely $\Omega(\frac{\log w}{\log\log w})$ and $\Omega(\frac{\log\log w}{\log\log\log w})$ respectively (his upper bounds are the same we give). He also does not allow double-precision operations. However, w is only a bound on the input size in [6]; the RAM may use unbounded words (and addresses) in computation. The results are thus incomparable.

It is remarkable that once weak non-uniformity is allowed, both of these functions can be computed in constant time and space. In fact, for an integer x, the position of the most significant non-zero bit of x is $\lfloor \log x \rfloor$; therefore, $\lceil \log x \rceil$ can be easily determined given a procedure for finding the most significant bit position. [8] gives a constant-time, constant-space procedure. $\lceil \log\log x \rceil$ is obtained by taking the logarithm twice.

Theorem 8. *A uniform RAM2 program requires the following time bounds for function computation:*

- $\Theta(\log w)$ *for computing* $\lceil \log x \rceil$, *provided* $a \leq w - w^\alpha$ *for some* $\alpha > 3/4$.
- $\Theta(\log\log w)$ *for computing* $\lceil \log\log x \rceil$, *provided* $a \leq w^\alpha$ *for some* $\alpha < 1$.

Proof. For both problems, we prove both lower bounds and matching upper bounds.

Lower Bounds: we prove both claims using Lemma 6. Note that for these problems, an instance consists of a single number, that is, $n = 1$. Thus the condition $w > (\log n)^{1/\delta} + 1$ is trivial. We assume $a > \log w$; if two machines are the same in all features except for the size of memory, the machine of smaller memory is obviously weaker; so the lower bound will hold for $a \leq \log w$ as a direct consequence.

For the first claim, we define $\varepsilon = \alpha$, $\gamma = 2\varepsilon - 1$ and $\delta = 2\gamma - 1$. Let $S = \{\, 2^i : p_0 \leq i < q_0 \,\}$
S is clearly external and contained in $N(l_0, p_0, q_0)$. S is distinguishable, for $\log 2^i = i$, a number of at most $1 + \log w \leq a$ bits. $|S| = q_0 - p_0 = \lfloor w^\delta \rfloor$; so $\log|S| = \delta \log w - o(1)$. The lemma yields a lower bound of $\Omega(\min(t, \log|S|)) = \Omega(\log w)$, since $t = (1 - \varepsilon)\log w$.

For the second claim, define $\varepsilon = \frac{7+\alpha}{8}$, $\gamma = 2\varepsilon - 1$ and $\delta = 2\gamma - 1$. Note that $\delta = \frac{1+\alpha}{2} > \alpha$. Let

$$S = \{\, 2^{2^i} : \alpha \log w \leq i < (\delta \log w) - 1 \,\}.$$

$i \leftarrow 1$
$l \leftarrow 0$
while $x > i$
    $i \leftarrow 2i$
    $l \leftarrow l + 1$
end while

Figure 6.1

$i \leftarrow 2$
$l \leftarrow 0$
while $x > i$
    $i \leftarrow i^2$
    $l \leftarrow l + 1$
end while

Figure 6.2

Each member of S contains a single 1-bit, whose position lies between w^α and $1/2 w^\delta$; it follows that S is external and contained in $N(l_0, p_0, q_0)$. For every $x \in S$, $\log\log x$ is a different integer, bounded by $(\delta \log w) - 1)$; the length of these integers is at most $1 + \log((\delta \log w) - 1) < 1 + \log\log w \leq a$ (we assumed $a > \log w$), so S is distinguishable. Finally, $|S| = (\delta - \alpha) \log w - 1$, so $\log |S| = \Omega(\log\log w)$. The lemma yields a lower bound of $\Omega(\min(t, \log |S|)) = \Omega(\log\log w)$, since $t = (1 - \varepsilon) \log w$.

Upper Bounds: Figures 6.1 and 6.2 give the claimed algorithms, whose validity and running time are straight-forward. The input variable is x and the output is given in l. $\square$

7. The Kirkpatrick-Reisch Algorithm

The RAM considered by Theorem 3 has a large memory and no double-precision multiply instruction. We claim the existence of a fast sorting algorithm in this model which is given by a uniform program, under a certain restriction on the word length. In this and the following sections, we describe algorithms that match these requirements: in the current section, we give an implementation of the Kirkpatrick-Reisch sorting algorithm [9] (called, in abbreviation, KR-sort), which runs in $O(n \log w)$ time; therefore it is fast if $\log w = o(\log n)$, and proves Theorem 4 for word lengths in this range. In the next section, we describe Modified Fusion Trees, which complete the proof for longer words.

For describing both algorithms, we adopt certain simplifications. First, we make use of non-uniform constants. To obtain a uniform algorithm, these constants should be constructed in a preprocessing stage in the manner shown in Section 2. Second, we assume that we have the largest memory, namely $a = w$. The case where a falls short of w by a small amount should be handled as in Section 3. Finally, we assume w to be a power of two. This assumption is fulfilled by all modern computers; however, the algorithms can be applied to other values with some technical effort.

The KR algorithm, as presented in [9], has the desired running time; however, it uses instructions like integer division, so in our presentation we make sure to demonstrate that it can be implemented with the basic instruction set.

Note first that the simplest fast sorting algorithm, bucket sort, suffices when $w = \log n$. To handle a larger word length, we use the following tool: an algorithm ALG, applicable to words of size w, can be adapted to sort words of a larger size w' on the leftmost slice of w bits (sorting a list on a certain slice means that the values encoded in this slice alone determine the ordering). For example, applying the standard *radix sort* (with radices of

$\log n$ bits) to words of size $w' = k \log n$ calls for performing k passes, such that in each pass *bucket sort* is applied to a slice of $\log n$ bits. In fact, it suffices to have a version of *bucket sort* which applies to the leftmost slice, because each of the other slices can be shifted to the left by multiplication. Below we use the example of *bucket sort* to illustrate the general technique. In general, we assume that ALG creates the sorted vector by transportation of words from the input vector and not indirectly by computing arithmetic expressions. Such an algorithm may be used to sort a vector according to any "sort key," for example, a part of the word.

> *Example*: ALG is *bucket sort*, for $w = \log n$. When applied to the vector $A[0 \ldots n]$, it puts $A[0]$, using indirect addressing, on a bucket which starts in the memory cell $\langle A[0] \rangle$; it proceeds to put $A[i]$ $(i = 1, \ldots, n-1)$ on buckets starting at $\langle A[i] \rangle$. Because $w = \log n$, the start addresses of buckets are in the range $[0, n-1]$. The algorithm concatenates the buckets in sorted order by accessing cell $\langle j \rangle$ for $j = 0, 1, \ldots, n-1$.

To adapt ALG to sorting on a left-justified slice (obtaining ALG'), some constants of the algorithm are multiplied by $2^{w'-w}$, thus justifying them within the slice (these will be the constants that should be commensurate with the sorted data). All computations are modified to apply to the leftmost w bits by ANDing the affected words with a mask M containing 1's exactly in the leftmost $w' - w$ positions. An execution of the modified algorithm, ALG', parallels exactly the execution of ALG except that all values used in computations are shifted by $w' - w$ bits. Hence, ALG' will sort correctly according to slice values.

> *Example (continued)*: Now A is an array of words of size $w' > w = \log n$. Let M be the mask for the leftmost w bits. Thus $A[i] \wedge M$ keeps the slice of w bits and clears the rightmost $w' - w$ bits. ALG' begins with a loop in which for $i = 0, 1, \ldots, n-1$, $A[i]$ is put on a bucket whose address is $\langle A[i] \wedge M \rangle$. Thus the start addresses of buckets are $0, \ldots, j \cdot 2^{w'-w}, \ldots, (n-1) \cdot 2^{w'-w}$. The algorithm concatenates the buckets in sorted order by accessing $\langle j \cdot 2^{w'-w} \rangle$ for $j = 0, 1, \ldots, n-1$. Note that these addresses span a large address space, although sparsely, confirming the need for a large memory.

The Kirkpatrick-Reisch algorithm relies on simple bucket-sort for sorting words (or slices) of $\log n$ bits or less. It operates on words of length $w > \log n$ by dividing them in two halves of $w/2$ bits each, and reducing the problem in linear time to that of sorting n halfwords. The method is used recursively to reduce the length of the slices that have to be sorted down to $\log n$, where bucket sort is usable. It should be clear from this description that the running time of the algorithm is $O(n \log(\frac{w}{\log n})) = O(n \log w)$. We describe the reduction from w to $w/2$. The following rounds differ only by the fact that they are applied to left-justified slices of $w/2$ bits, $w/4$ bits etc, and therefore require the following constants for accessing the respective slices: $2^{\frac{w}{2}}, 2^{\frac{3w}{4}}, 2^{\frac{7w}{8}}$ etc.

We now describe a reduction round, whose purpose is to reduce the problem on w bits to a sorting problem on the $w/2$ leftmost bits. For each input number I_k we consider the pair (x_k, y_k) of the upper and lower halfwords of I_k; note that both values can be obtained in the upper halfword, setting $x_k = I_k \wedge (-2^{w/2})$, $y_k = I_k \cdot 2^{w/2}$. The numbers are now assigned to buckets according to their upper half: I_k is assigned to the bucket $B[x_k]$. To do this efficiently, bucket $B[i]$ is represented as a linked list whose head is at memory address i; this way each addition to a bucket takes constant time. While there are $2^{w/2}$ potential buckets, at most n will be actually used, so we also maintain a list of Active-Buckets. Each bucket which is used for the first time is added to the list.

It is easy to see that everything will be sorted properly if we accomplish the following two tasks: (i) sort Active-Buckets; (ii) sort each bucket by the second coordinate. In order to obtain the desired running time, we want to call the algorithm recursively only once with a list of n data. Let n_b be the number of buckets used. We remove from each bucket the maximal element (finding all these maxima takes linear time). This reduces the number of elements in the buckets by n_b; we now concatenate the contents of all the buckets plus the Active-Buckets list to a single list of n pairs (u, v) defined as follows. For each member of Active-Buckets, u is the name of the bucket and $v = 1$. For each element I_k $(\in B[x_k])$, $u = y_k$ and $v = x_k$. Note that x_k has zeros in the lower halfword, so in particular it cannot equal 1. Thus pairs representing bucket elements are distinguishable from pairs representing active buckets. The list is sorted on u (by the recursive call). Afterwards, we split the list, putting each bucket element back in its bucket, and each bucket name on Active-Buckets. Now tasks (i) and (ii) are accomplished and the algorithm is easily completed.

8. Modified Fusion Trees

In this section we complete the proof of Theorem 4, by presenting the Modified Fusion Tree algorithm (MFT).

The only algorithm we know, that is fast for larger words than KR sort can handle, is the Fusion Tree algorithm. But this algorithm uses double-precision multiplication, which we exclude from our model. This instruction is used essentially for moving certain bits from a given word to less significant positions in another word. It is not possible to simulate this operation efficiently in our model, unless substantially restricted. The next lemma gives the favorable case.

Lemma 7. *On a RAM with large memory it is possible to implement an operation that moves $t \le \log n$ bits from the left end of a word to the right end in constant time, after $O(n)$-time preprocessing.*

Proof. In the preprocessing stage we build a table SHIFT of 2^t entries, indexed by the numbers $i \cdot 2^{w-t}$, $\quad 0 \le i < 2^t$ (this is a very sparse table which exploits the large memory). For each i, we store i in SHIFT$[i \cdot 2^{w-t}]$. This takes $O(2^t)$ operations and completes the preprocessing. Now, to set y to the most significant t bits of x, we simply execute

$$y \leftarrow \mathrm{SHIFT}[x \wedge (-2^{w-t})] \qquad \square$$

In fact we can do more: the range of t bits starting at an arbitrary bit k can be retrieved (that is, right-justified within a word) provided we have the constant $C = 2^{w-(t+k)}$; we simply multiply by C and apply SHIFT. Furthermore, t may be a constant multiple of

$\log n$; we move $\log n$ bits at a time. Below we use the expression "move a certain range of bits using SHIFT" in reference to this procedure.

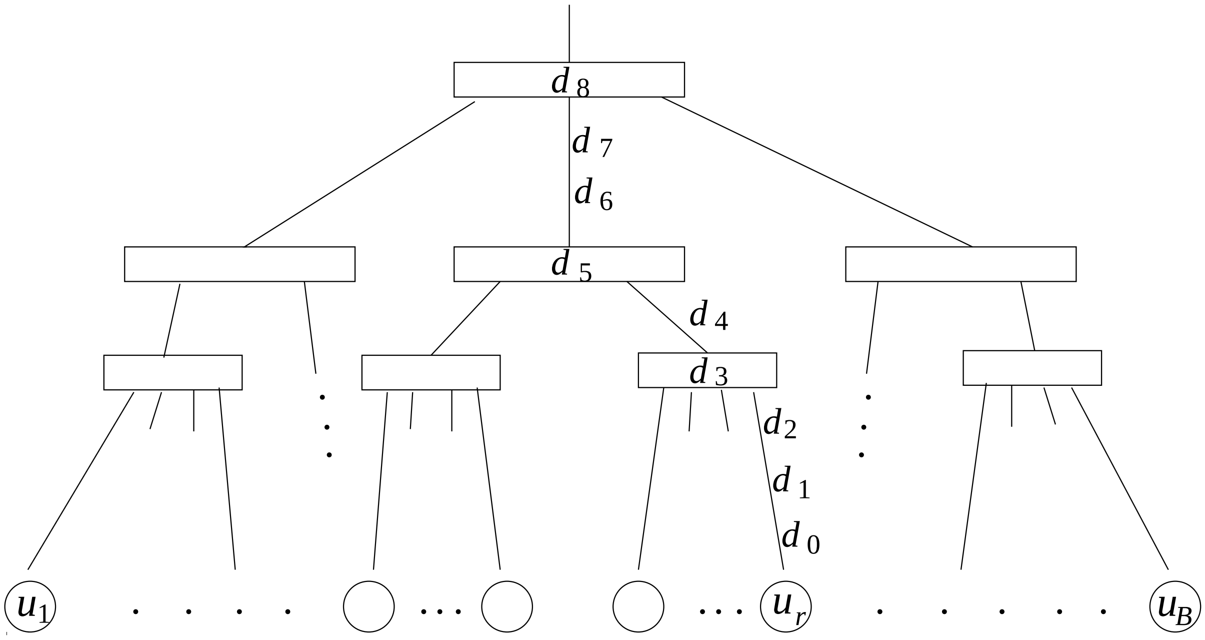
The Fusion Tree algorithm has to be modified so that all uses of double-precision operations are eliminated, except for the operation covered by Lemma 7. The modifications yield a fast algorithm provided $\log\log\log w = o(\log n)$. However, in order that the construction of non-uniform constants be fast enough, we need $w = \dot{o}(2^{n \log n})$. Therefore, in the following description we will assume this stronger condition, which simplifies the algorithm somewhat.

Throughout this section, we adopt the convention that positions within a word are counted from right (less significant) to left (more significant). Expressions like "preceding" or "succeeding" regarding parts of a word make use of this convention. The symbol $\triangle$ is used for the bitwise exclusive or (XOR) operation.

Let $s = 2^{\lfloor \sqrt{\log n} \rfloor}$. If $w \le s$, the Kirkpatrick-Reisch algorithm is fast. Modified Fusion Trees are only used if $w > s$. In this case, we use a two-pass radix sort in which KR is applied to the high-order slice of s bits, and Modified Fusion Trees to the rest of the word. Hence the MFT algorithm will assume the high-order s bits to be all zero. As discussed above, all that we need to implement this strategy is to obtain, fast enough, the constants s and 2^{w-s}. It is easy to see that using repeated doubling/squaring techniques, s can be computed in $O(\sqrt{\log n})$ time, and 2^{w-s} in $O(\log(w-s)) = o(n \log n)$ time.

8.1 The Data Structure

A B-tree [3] is a search tree where each internal node has at most B children, for some $B \ge 3$. The algorithms of [3] maintain the tree so that each node has at least $B/2$ children, and a tree of n leaves has height $O(\log_B n)$. A node of degree k is associated with $k-1$ keys, used as search discriminators. When searching for a value u, we determine the child where the search should proceed by comparing u with these keys. Sorting n elements using such a tree requires n node updates and n key searches. The Modified Fusion Tree (MFT) is such a tree. We will choose $B \approx \sqrt[6]{\log n}$ (actually B will be the largest power of two bounded by $\sqrt[6]{\log n}$). The data structure that represents a node, and the algorithms to build and search nodes, are quite intricate and occupy most of this section. They achieve the following performance: when performing a key search, the correct child of each node along the search path can be determined in constant time. Hence the n searches will take $O(n \log_B n) = O(n \frac{\log n}{\log\log n})$ time. Each update to a node will be performed by rebuilding the node. We implement this operation in $O(B^4) = O((\log n)^{4/6})$ time. Therefore the total time to sort will be $O(n \frac{\log n}{\log\log n})$.



In this example $m = 9$; the distinguishing digits are d_8, d_5 and d_3.

Figure 8.1

8.2 Rank Computation with Compressed Keys

In order to search through each level of the tree in constant time, we want a node to support in constant time the following operation. let $S = \{u_1 < u_2 < \cdots < u_B\}$ be the set of keys at this node (actually less keys are stored, but is simplifies the description to round the number up to B). Given an arbitrary value u, we wish to compute $\mathrm{rank}_S(u)$, the number of elements in S smaller than u. Extending the idea used in Fusion Trees (which goes back to [2]), we now describe a way of computing the rank with the help of *compressed keys*. This idea is strongly related to the *trie* data structure, a search tree in which each internal node branches according to the value of a certain digit of the key.

Let $b_1, b_2, \ldots, b_m$ be integers such that $\sum_{i=1}^{m} b_i = w$. These numbers define a subdivision of the binary word into *fields*, where field i has b_i bits. Another way to view these fields is as digits in a mixed-radix system, with radices 2^{b_i}. In this system, each number up to $2^w - 1$ is represented by m digits. Digits are numbered from least significant (position 0) to most significant (position $m-1$), and accordingly we write a number as $d_{m-1} \ldots d_1 d_0$.

Definition 1. *Let $0 \le u \ne v < 2^w$. The distinguishing position $DP(u, v)$ is the largest j such that u differs from v on d_j.*

$DP(u, v)$ is the position of the most significant digit that distinguishes u from v.

A standard trie structure has m levels, each level corresponding to a digit position. The first level corresponds to the most significant digit (d_{m-1}), the second to the next one (d_{m-2}) and so forth. We consider a trie that represents the set S. For $1 \le i < B$, let

$c_i = DP(u_i, u_{i+1})$. The paths followed in the trie by u_i and u_{i+1} diverge exactly in the level c_i. Since only distinguishing positions induce branching in the trie, we adopt a version of this structure in which only the levels that correspond to distinguishing digits are kept. Each arc corresponds to a sequence of (zero or more) digits which are not distinguishing (Figure 8.1).

We now describe a procedure for finding $\mathrm{rank}_S(u)$ for an arbitrary $0 \le u < 2^w$. The trie structure motivates our algorithm, but is not explicitly maintained. The reader should bear in mind that it is only used to explain the significance of the values we compute.

Let $D(S) = \{c_1, c_2, \ldots, c_{B-1}\}$, the set of distinguishing positions. Let $k = |D(S)|$.

Definition 2. *For an arbitrary $0 \le u < 2^w$, the compressed value $\hat{u}$ is the k-digit number obtained by picking up from the digits of u just those in distinguishing positions.*

For example: in Figure 8.1, $\hat{u}_r = d_8 d_5 d_3$.

The first step of the rank computation is to find the rank of $\hat{u}$ relative to the set of *compressed keys* $\hat{S} = \{\hat{u}_1, \ldots, \hat{u}_B\}$ (note that compressed keys retain the increasing order). We do not give, in this subsection, an implementation of this operation, neither do we show how to find distinguishing positions or how to compress a number. These operations should be treated as subprocedures whose implementation is not part of the algorithm currently described. Regarding complexity, we show that the time for finding $\mathrm{rank}_S(u)$ is constant if these subprocedures take constant time individually. Modified Fusion Trees use elaborate algorithms for achieving this result, which are described in later subsections.

So, let $\hat{\rho} = \mathrm{rank}_{\hat{S}}(\hat{u})$. If $u \in S$, it will equal $u_{\hat{\rho}+1}$. So we next check this condition. If it does not hold, we know that u differs from all the keys of S. Tracing the digits of u through the trie for S, there will be a first position p where it does not agree with any of the keys. We distinguish two cases, according to whether this position is distinguishing or not. The first case is demonstrated by u' in Figure 8.2. It is easy to verify that in this case, $\mathrm{rank}_S(u) = \hat{\rho}$. So our third step is to check whether $u_{\hat{\rho}} < u < u_{\hat{\rho}+1}$. If this holds, we know the rank of u; otherwise, we know that we have the second case, where position p corresponds to an arc of the trie (see u'' in Figure 8.2, where $p = 7$). Let u_l (respectively u_r) be the leftmost (rightmost) leaf in the subtree that descends from the arc where u diverges from the trie (see figure). The rank of u is either r (when u has a larger value of d_p, as the figure illustrates for u'') or $l-1$ (in the symmetric case). Therefore, if we can determine l and r, a single comparison will yield the rank of u.

Observe that $\hat{u}$ starts with some distinguishing digits in which u agrees with the trie (d_8 in our example). These are the digits before position p; after that point, the values of the distinguishing digits in u may bear any relation to those of the other keys, with no effect on $\mathrm{rank}_S(u)$. Recall that $\hat{\rho}$ is determined by the compressed keys, i.e., only by digits in distinguishing positions; hence $\hat{\rho}$ will be between $l-1$ and r. In the fourth step of the rank procedure we establish a value $\bar{\rho}$ such that $l \le \bar{\rho} \le r$. Naturally, this will be either $\hat{\rho}$ or $\hat{\rho}+1$. If $\hat{\rho}$ is 0 or B, we take $\bar{\rho}$ to be 1 or B, respectively. Otherwise $0 < \hat{\rho} < B$ and we proceed as follows. We find the positions that distinguish u from $u_{\hat{\rho}}$ and $u_{\hat{\rho}+1}$. Call these positions $p_{\hat{\rho}}$ and $p_{\hat{\rho}+1}$. The following cases are possible: if $\hat{\rho} = l-1$, $p_{\hat{\rho}} > p$, and $p_{\hat{\rho}+1} = p$. If $\hat{\rho} = r$, $p_{\hat{\rho}+1} > p$, and $p_{\hat{\rho}} = p$; in all the intermediate cases, $p_{\hat{\rho}} = p_{\hat{\rho}+1} = p$. Therefore, we

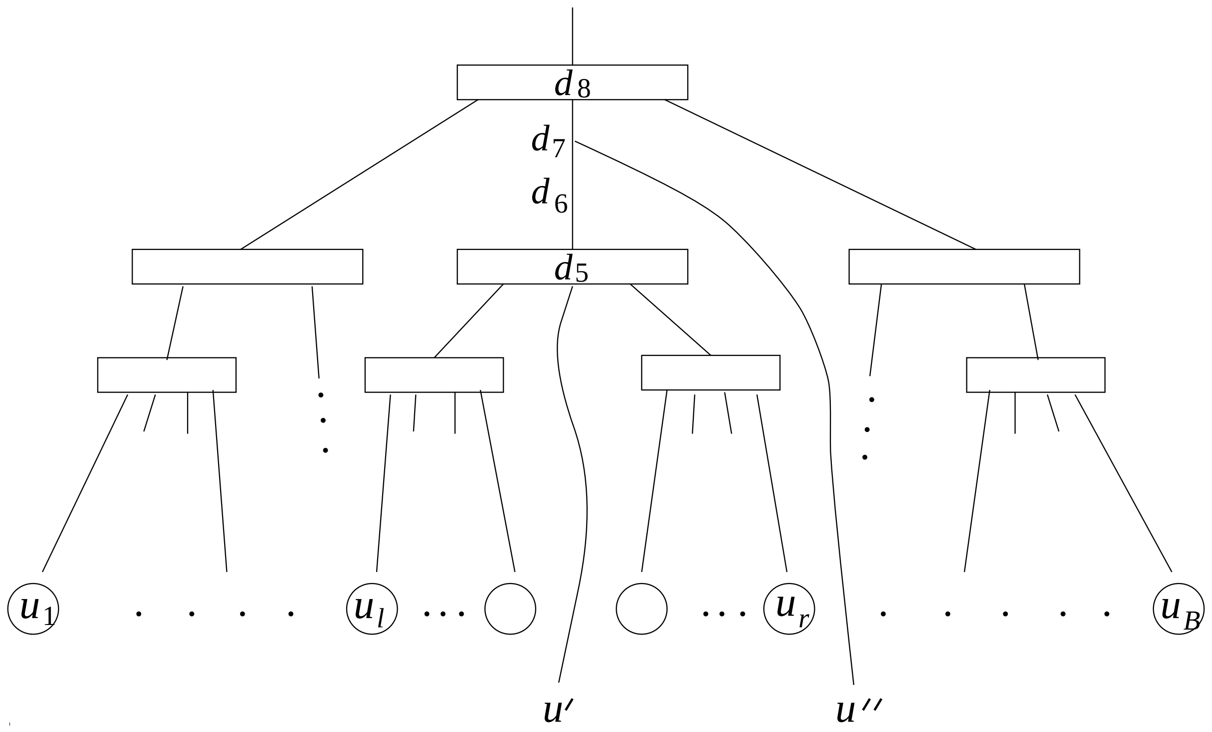


Figure 8.2

compare $p_{\hat{\rho}}$ to $p_{\hat{\rho}+1}$: if $p_{\hat{\rho}} < p_{\hat{\rho}+1}$, then $\hat{\rho}$ cannot be $l-1$, and we choose $\bar{\rho} = \hat{\rho}$. Otherwise, $\hat{\rho}$ cannot be r; we choose $\bar{\rho} = \hat{\rho}+1$.

We now have a value $\bar{\rho}$ that is guaranteed to be between l and r. Recall that $c_i = DP(u_i, u_{i+1})$ is the position where the paths to u_i and u_{i+1} diverge. It is easily established that

$$l = \max\{i \le \bar{\rho} : c_{i-1} > p\}$$
$$r = \min\{i \ge \bar{\rho} : c_i > p\}$$

(we define $c_0 = c_B = m$).

We conclude that, for a given set of keys, we only need the following data in order to find $\mathrm{rank}_S(u)$: $\bar{\rho}$ (which can be determined from $\hat{\rho}$, $p_{\hat{\rho}}$ and $p_{\hat{\rho}+1}$), and the *rank of* p *relative to* $D(S)$ (this rank determines which c_i are greater than p). Our algorithm will use a table, indexed by the values of $\bar{\rho}$ and this rank, to retrieve l and r. This table will be built as part of the construction of a node from a set of keys. First, we will find the distinguishing positions for the keys. Given $\{c_i\}$, constructing our table takes $O(B^2)$ time. After this processing, $\mathrm{rank}_S(u)$ is computed, for any given u, by performing the following steps until the rank is found.

1. If $u > u_B$ then $\mathrm{rank}_S(u) = B$. If $u \le u_1$ then $\mathrm{rank}_S(u) = 0$.
2. Let $\hat{\rho} = \mathrm{rank}_{\hat{S}}(\hat{u})$.
3. If $0 < \hat{\rho} < B$, and $u_{\hat{\rho}} < u \le u_{\hat{\rho}+1}$, then $\mathrm{rank}_S(u) = \hat{\rho}$.
4. If $\hat{\rho} = B$, let $\bar{\rho} = B$; if $\hat{\rho} = 0$, let $\bar{\rho} = 1$; if any of these conditions holds, skip to Step 6.

5. Let $p_{\hat{\rho}} = DP(u, u_{\hat{\rho}})$, and $p_{\hat{\rho}+1} = DP(u, u_{\hat{\rho}+1})$. If $p_{\hat{\rho}} < p_{\hat{\rho}+1}$, let $\bar{\rho} = \hat{\rho}$; otherwise let $\bar{\rho} = \hat{\rho} + 1$.

6. Let $p = DP(u, u_{\bar{\rho}})$; use $\bar{\rho}$ and $\text{rank}_{D(S)}(p)$ as indices to a table that includes a pair l, r.

7. If $u \le u_l$ then $\text{rank}_S(u) = l - 1$; otherwise $\text{rank}_S(u) = r$.

Remark 1. In our the implementation of compression, a compressed key will be a bit string that contains the required fields (those in $D(S)$) in fixed positions, and may be padded with zeros between them. This allows for faster computation, and clearly preserves their order relations.

Remark 2. Fredman and Willard's Fusion Trees use a special case of this method, in which each bit of the word serves as a digit. Their trie is thus binary.

8.3 Node Representation

We choose B, the node degree, to be the largest power of two bounded by $\sqrt[6]{\log n}$ (if $B < 3$, n is too small and we do not use this algorithm). Let $q = \log w - \log s + 2$, so $\frac{w}{2^{q-2}} = s$. Let $N = 16B^5$ and $F = \frac{w}{2^q NB}$.

Lemma 8. *For n large enough, the following hold:*

(i) F *is a positive power of two.*

(ii) $\log qN < F$.

(iii) $\log q < \frac{1}{16} F$.

Proof.

(i)
$$F = \frac{w}{2^q NB} = \frac{s}{4 \cdot 16B^6} \ge \frac{2^{\lfloor \sqrt{\log n} \rfloor}}{2^6 (\log n)}$$

From this it follows that $F > 1$ for n large enough. Since both the nominator and denominator are powers of two, F will be a positive power of two.

(ii) and (iii) both follow from the fact

$$\log q < \log qN = O(\log n) = o(F)$$

which we prove in two parts: first,

$$\log(qN) < \log((\log w)N) \le \log((\log w) \cdot 16(\log n)^{5/6}) \le \log((\log w)(\log n)) \, .$$

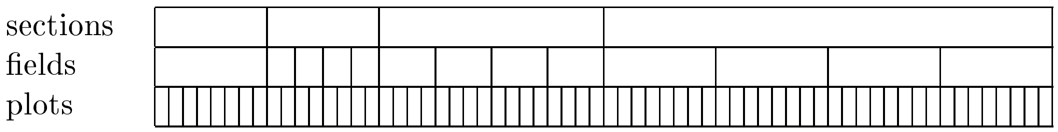
Since $w = \dot{o}(2^{n \log n})$, we may assume $\log w < n \log n$. Hence

$$\log((\log w)(\log n)) < \log(n(\log n)^2) = \log n + 2 \log \log n = O(\log n) \, .$$

Secondly,

$$\frac{\log n}{F} \ge \frac{2^6 (\log n)^2}{2^{\lfloor \sqrt{\log n} \rfloor}} \longrightarrow 0$$ □

We define three levels of subdivision on the word: *sections*, *fields* and *plots* (see Figure 8.3; we remark that the values of q, N, B there are not really possible). Section i (recall



The subdivision of the word in case $q = 3$, $N = 4$ and $B = 2$.

Figure 8.3

that we count from the right) consists of $L_i = w/2^i$ bits. There are q sections. Each section is divided into N equal fields, whose size we denote by $f_i = \frac{w}{2^i N}$. The fields of the smallest section are of size $f_q = BF$, and those of larger sections have sizes which are multiples of BF. *Plots* are simply slices of F bits. The leftmost $s/4$ bits do not belong to any section, and are called *the buffer area.* Since Modified Fusion Trees assume the high-order s bits to be zero, the buffer area is constantly zero in our input (as are the last two sections).

As described in the last section, each field plays the role of a digit in the mixed-radix representation. A compressed key will include a certain subset of the fields, the distinguishing positions. To complete the description of the algorithm, we will show how, when building a node, we can locate the distinguishing fields and construct a compressed representation within $O(B^4)$ time; and how, given a new value, we can compute its rank in constant time.

8.4 Special Constants

The algorithm requires some constants, dependent on n and w, that should be computed in a preprocessing phase to make the algorithm uniform. These include q, B, N and F defined above, and all of the following list.

$K_1 = 2^{\frac{F}{B}}$.

$K_2 = 2^F - 1$.

$K_3 = 2^{w-\frac{s}{4}}$ (a 1 right-justified within the buffer area).

M_1 contains ones in the upper quarter of each even-numbered section.

M_2 contains ones in the upper quarter of each odd-numbered section.

M_3 contains the value 1 in every field.

M_4 contains the value 1 in every plot.

M_5 contains a left-justified 1 every $4B^4$ fields.

$M_6 = \mathrm{bin}(f_1, f_2, \ldots, f_q, 0)$.

$M_7 = \mathrm{bin}(L_1, L_2, \ldots, L_q, 0)$.

$M_8 = \mathrm{bin}(B^4 f_1, B^4 f_2, \ldots, B^4 f_q, 0)$.

$M_9 = \mathrm{bin}(L_1 - L_q, L_2 - L_q, \ldots, L_{q-1} - L_q, 0)$.

$M_{10} = \mathrm{bin}(0, F, 2F, \ldots, qF)$.

Procedures for efficient construction of similar constants have been presented in Section 2. Their running time is $O(\log w) = o(n \log n)$.

8.5 Some Definitions and Programming Tricks

Recall that the role of *digits* is played in our structure by *fields*, while in Fredman and Willard's algorithm, each digit is a single bit. It is more convenient to think of operations in terms of digits which are single bits, and we sometimes start with operations which are well known for the case of bits, and extend them to fields and sections.

There are two kinds of operations on bits: *masking* and *moving*. Masking is used to *select* a subset of the bit positions, using a mask that contains a 1 in each selected position. We sometimes say that we *clear* the unnecessary positions (this is exactly the same operation).

We will need masks to select a particular field. We denote by $F(j)$ a mask that selects field j (where $0 \le j < qN$ is the field position within the word, counting from the right, beginning at 0).

Moving bits within the word is accomplished with the multiplication operation. Let $\text{bin}(i_1, i_2, \ldots) = \sum_j 2^{i_j}$, namely an integer where bits i_j (and none else) are set. Multiplying x by $\text{bin}(i)$ is tantamount to shifting it i positions to the left; multiplying by $\text{bin}(i_1, i_2, \ldots)$ amounts to adding together the (i_j)-shifted replicas of x. If the (i_j) are chosen carefully, there will be no carries in this addition, in other words: the replicas will not collide. In this case we have *a clean multiplier*. This operation is a key tool in our algorithm, and the reader may find it convenient to regard $\text{bin}(i)$ as a *shift operator*, and a many-bit multiplier as a collection of shift operators which are applied simultaneously.

When a digit is a bit, we build our multipliers to move selected bits to a desired distance, also measured in bits. To handle fields, we simply have to scale the distances to the appropriate field size.

Let $x = \text{bin}(i_0, i_1, \ldots, i_t)$ where $0 = i_0 < i_1 < \cdots < i_t < w - 1$. Such a word defines a subdivision of the word into slices, such that slice j includes bit positions $i_j, \ldots, i_{j+1} - 1$ (bits $i_t, \ldots, w-1$ constitute slice t). In x, every slice contains the value 1. We define $\text{lead}(z, x)$, for an arbitrary word z, to be a word in which slice $j+1$ contains a 1 whenever slice j of z is non-zero, and a 0 otherwise. For example:

x	1	1	1	1	1	1	1
z	00	00	11	0101	0000	1000	0000
$\text{lead}(z, x)$	0	1	1	0	1	0	0

$\text{lead}(z, x)$ can be computed in constant time using the following identity.

Lemma 9. $\text{lead}(z, x) = \big((\neg x) + (\neg x \wedge z)\big) \vee ((\neg x) + 2(x \wedge z)) \wedge x.$

Proof. We use the notation $[x]_k$ for bit k of x. We prove below the following two claims.

(1) $[(\neg x) + (\neg x \wedge z)]_{i_{j+1}} = 1$ if and only if bits $i_j + 1, \ldots, i_{j+1} - 1$ of z are not all zero.

(2) $[(\neg x) + 2(x \wedge z)]_{i_{j+1}} = 1$ if and only if $[z]_{i_j} = 1$.

Together, the claims imply that bit i_{j+1} of (1)$\vee$(2) is 1 if and only if slice j of z is non-zero. The AND with x clears all other bits, obtaining $\text{lead}(z, x)$.

Proof of (1). $\neg x$ contains in slice j the value $11\ldots10$. $\neg x \wedge z$ keeps in slice j bits $i_j+1,\ldots,i_{j+1}-1$ of z. Both $\neg x$ and $\neg x \wedge z$ have bit i_{j+1} off. In adding them, this bit may be set only as the result of a carry. Since bit i_j is zero in both numbers as well, carries do not propagate from preceding slices; the carry into bit i_{j+1} is due solely to adding slice j of $\neg x$ to slice j of $\neg x \wedge z$. It will be a 1 if and only if bits $i_j+1,\ldots,i_{j+1}-1$ of z are not all zero.

Proof of (2). $\neg x$ contains in slice j the value $11\ldots10$. $x \wedge z$ contains a 1 only in positions i_j such that $[z]_{i_j}=1$. $2(x\wedge z)$ contains a 1 in position i_j+1 if and only if $[z]_{i_j}=1$. Consequently in adding this value to $\neg x$, a carry will appear in bit i_{j+1} if and only if $[z]_{i_j}=1$. $\square$

Lemma 10. *Let $x=\mathrm{bin}(i_0,i_1,\ldots,i_t)$ where $\log t<F$, $0=i_0<i_1<\cdots<i_t<w-1$ and each i_j is a multiple of F. Given x and $0\le k\le t$, we can obtain $\mathrm{bin}(i_k)$ in constant time.*

Proof. If $k=0$, $\mathrm{bin}(i_k)=1$. We continue for $k>0$. Compute $y=M_4\cdot x$. It is easy to verify that y contains the value $j+1$ in every plot of slice j. We are making use of the fact that $\log t<F$, so there is no overflow from the plot. Set $z=y\triangle(M_4\cdot k)$. z will contain 0 in a plot if and only if this plot is in slice $(k-1)$. Compute $u=x-\mathrm{lead}(z,x)$. This word contains a 1 in slice k, and 0 elsewhere. That is, $u=\mathrm{bin}(i_k)$. $\square$

Lemma 11. *Using our special constants, we can compute the following in constant time: given $1\le i\le q$, the values 2^{f_i}, 2^{L_i}, $2^{B^4 f_i}$, $2^{L_i-L_q}$, 2^{iF}. Given $0\le i<Nq$, the mask $F(i)$.*

Proof. The first five values are obtained by applying Lemma 10 to M_6 through M_{10}, respectively. For the last one, define $C(j)$ to be a word where field $j+1$ contains the value 1, and all other fields are zero. Given $j<Nq-1$, $C(j)$ can be computed from M_3 using the last lemma; $C(Nq-1)=K_3$. Now $F(i)=C(i)-C(i-1)$. $\square$

8.6 Locating Distinguishing Fields

In order to construct a node from a set S of (at most B) keys, we first have to sort the keys, which can be done with any $O(B^2)$-time algorithm. Then we have to locate the distinguishing field for each pair of consecutive keys. Consider two keys u and v. We next show how to find the position of their distinguishing field in constant time. It is easy to see from the definition of the distinguishing digit that it is in fact the most significant non-zero digit of $u\triangle v$. We thus need a procedure for computing $\mathrm{MSD}(x)$, the position of the most significant digit (i.e. field) of a given x. Given the position, we will also obtain a mask for the most significant field by Lemma 11.

The first step in the MSD procedure is computing $\mathrm{lead}(x,M_3)$, where field $j+1$ is set to 1 if and only if field j is different from zero. Let $y=M_4\cdot\mathrm{lead}(x,M_3)$. y contains in every plot the count of non-zero fields in x that lie to its right: this number is bounded by the number of fields, qN, and $\log qN<F$ (Lemma 8), so there is no overflow. The last plot, within the buffer area, contains the number of non-zero fields in x. We call this value $\mathrm{count}(x)$. Since $\log qN=O(\log n)$, we can right-justify $\mathrm{count}(x)$ using SHIFT.

Next, we clean y up by ANDing it with the word M_3K_2, thus clearing all but the rightmost plot of each field, eliminating repetitions within the field. y now contains in each field the number of non-zero fields to its right. Clearly these values increase leftward, culminating in $\mathrm{count}(x)$. The latter appears exactly in all fields that succeed the MSD (recall that the last section is constantly zero. Thus the MSD is not the leftmost field). Let $z = M_3 \cdot \mathrm{count}(x)$; z contains $\mathrm{count}(x)$ in every field. Let $t = z - y$. In t, every field left of the MSD contains a zero while all fields up to the MSD contain a non-zero value. We now compute $\mathrm{count}(t)$, the number of non-zero fields in t. $\mathrm{MSD}(x) = \mathrm{count}(t) - 1$.

During node construction, we use this procedure for computing the distinguishing position set $D(S)$. Since the procedure takes constant time, $D(S)$ is found in $O(B)$ time. During rank computation, we use this procedure for computing the distinguishing positions $p_{\hat{\rho}}$, $p_{\hat{\rho}+1}$ in constant time.

8.7 Rank of Distinguishing Position

In computing the rank of a new key, we are required to compute $\mathrm{rank}_{D(S)}(p)$, where p is the position of the distinguishing field for a certain pair of words, that is, the most significant field of their XOR. We saw above how, during node construction, we obtain the set $D(S)$. It is straightforward to create at the same time a mask W that has a 1 in field $j+1$ if and only if field j is distinguishing (for uniformity of handling, we set field 0 to 1 as well). This mask defines a subdivision of the word into slices, whose borders are the left ends of distinguishing fields.

Observe that for an arbitrary word, the position of its most significant non-zero slice equals the rank of its MSD relative to $D(S)$. Therefore, our task is accomplished by computing the index of the most significant non-zero slice. This procedure proceeds identically to the computation of the most significant field position (last subsection), except that W is used instead of M_3. As in the last subsection, this operation takes constant time.

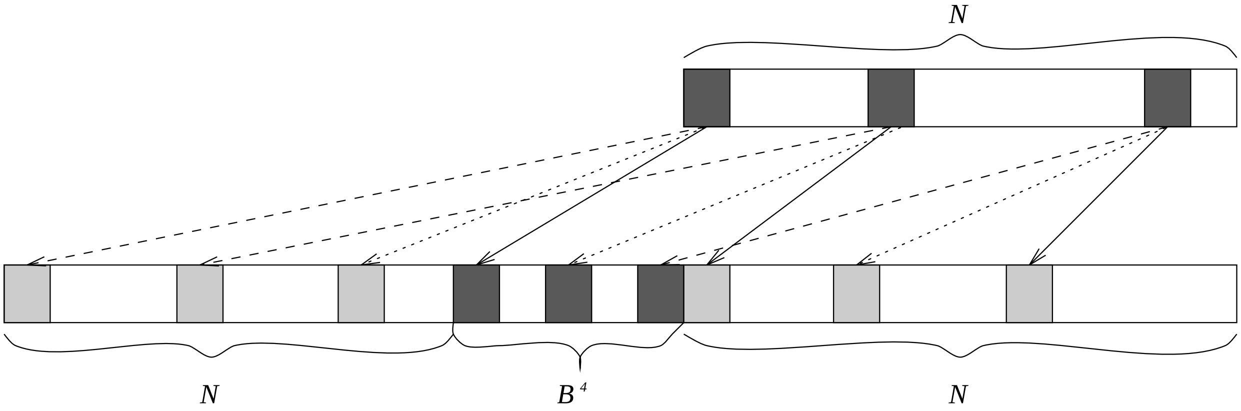
We now miss only one part of the rank algorithm of Section 8.2, and this is the part involving compressed keys: we need procedures for compressing keys, to produce $\hat{S}$ and $\hat{u}$, and for computing $\hat{\rho} = \mathrm{rank}_{\hat{S}}(\hat{u})$, given a new value u.

8.8 Compression

Recall that compression of a key u involves picking out from u the values of distinguishing digits. So after calculating the set $D(S)$ of distinguishing field positions for the set S of keys, we create the mask $C_S = \sum_{i \in D(S)} F(i)$. Using C_S, we clear all but the distinguishing fields of u.

Given the position p of a field within the word, $p \bmod N$ is the field's position inside its own section, and $\lfloor p/N \rfloor + 1$ is the section number. Since N is a power of two, computing $p \bmod N$ is easily accomplished using AND. Computing $\lfloor p/N \rfloor$ amounts to right-shifting p by $\log N$ bits, which can be done with SHIFT since the number of bits in p is bounded by $\log qN$. So we compute

$$B(S) = D(S) \bmod N = \{p \bmod n : p \in D(S)\}$$



The dark grey blocks represent the distinguishing fields.
The light ones are side outcomes of the multiplication.
The multiplier contains 3 one-bits, corresponding to solid, dotted and dashed arrows resp.

Figure 8.4

and

$$\bar{B}(S) = \{\lfloor p/N \rfloor + 1 : p \in D(S)\}$$

$B(S)$ is a set of distinguishing field positions, detached from their section numbers. $\bar{B}(S)$ is the set of sections that contain distinguishing fields (called *distinguishing sections*). To simplify discussion, we consider in most of what follows a single, representative section, called section σ, and the distinguishing fields within it (which may happen to be the whole set). The algorithm will apply the procedures that we describe to all sections concurrently, exploiting the bit-level parallelism of arithmetic instructions. Thus all distinguishing fields will be treated, each within its own section.

We denote the contents of our representative section (already cleaned up using C_S) by Σ and its section number by σ. The term "field" will refer henceforth to the field size of this section, and we view the *whole word* (even outside of our section) as divided into fields of size f_σ. This attitude is taken because when operating on section σ, we move some of its fields into parts of the word that originally belong to other sections.

The number of distinguishing fields is bounded by B. Compression calls for relocating these fields into a small enough part of the word: in particular, small enough relative to the size of a section, which is $N = 16B^5$ fields. This relocation will be carried out using a set of multipliers and masks. We will describe the process first, defining the multipliers and masks used, and argue its correctness; later we describe how these numbers are obtained.

The following lemma is essentially proved in [8].

Lemma 12. *Given $D(S)$, a clean multiplier m_S can be found such that multiplying Σ by m_S will relocate its distinguishing fields into a range of up to B^4 contiguous fields, preserving their order. The product $\Sigma \cdot m_S$ occupies up to $2N + B^4$ fields with the range of interest in the middle (Figure 8.4).*

We define $\hat{u}_\sigma$ to be the resulting range of B^4 fields, which contains the distinguishing fields. All other bits of the result are cleared to zero by an appropriate mask. We refer to $\hat{u}_\sigma$ as a

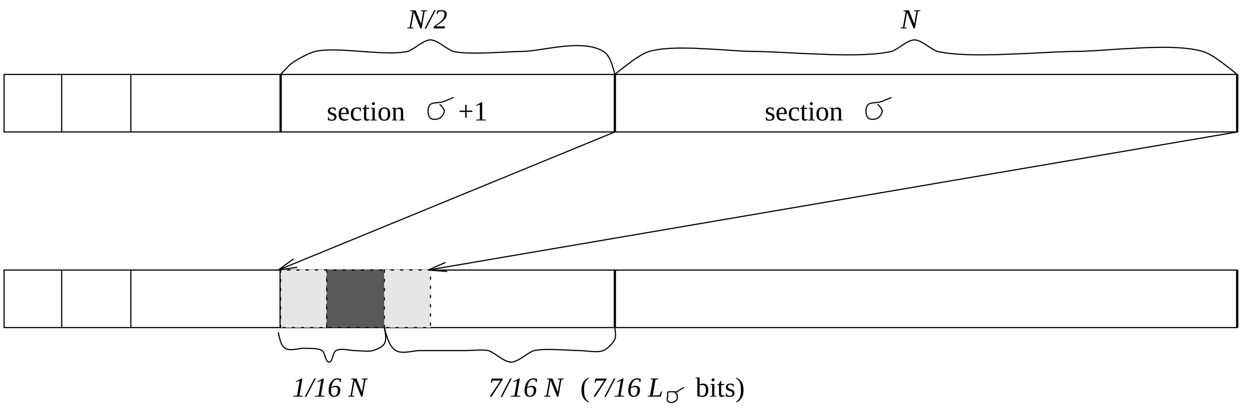


Figure 8.5

compressed block. Since we clear non-distinguishing fields first with the mask C_S, sections that contain no distinguishing fields will yield zero compressed parts and we can write

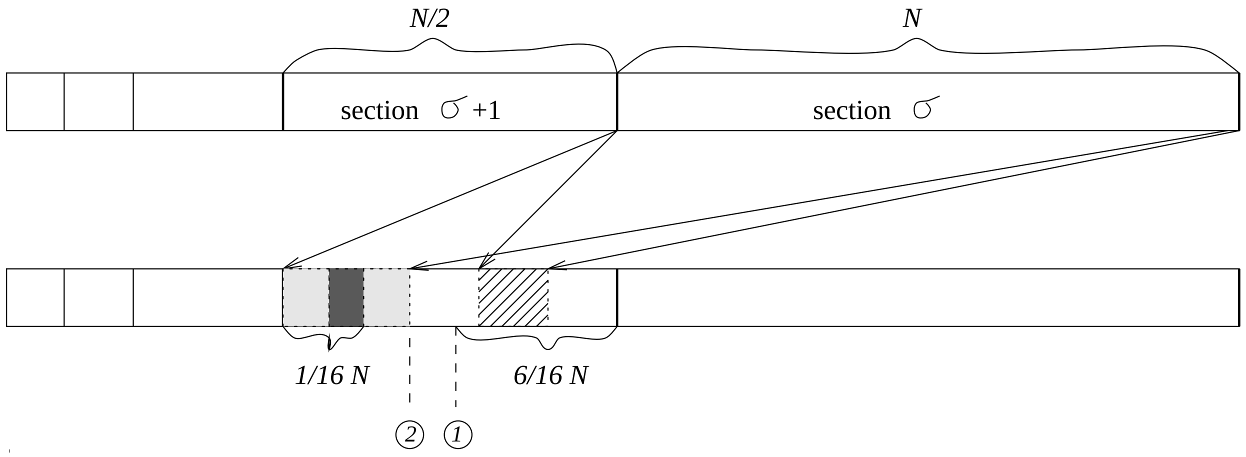
$$\hat{u} = \hat{u}_{\sigma_k} \hat{u}_{\sigma_{k-1}} \ldots \hat{u}_{\sigma_1} \qquad \text{where} \qquad \{\sigma_k > \sigma_{k-1} > \cdots > \sigma_1\} = \bar{B}(S) \ .$$

Compressing all sections in parallel is essentially simple. Recall that a multiplier is a collection of shift operators. Find the shifts, described by the above lemma, for all the distinguishing fields, and scale each one to the field size of the respective section. Using a multiplier which includes all these shifts, a single multiplication will compress all sections. However, two complications arise. The first is the following: the multiplier of the lemma is clean when applied to a single section. When the whole word is considered, we find that the side outcomes on both sides of the compressed block (Figure 8.4) overlap neighboring sections, making the combined multiplier unclean.

The problem is handled by dividing m_S into 32 pieces, $m_S^0, \ldots, m_S^{31}$, such that $\sum_{i=0}^{31} m_S^i = m_S$, but if we multiply Σ by a single piece the product only occupies $\frac{1}{16}N + B^4$ fields, $\frac{1}{32}N$ on each side of the compressed block. Recall that $N/2$ is the length (measured in "our" fields) of the next section $(\sigma + 1)$. Since $\frac{1}{32}N + B^4 \leq N/16$, we can adjust m_S so that the compressed block is right-justified in the last eighth ($N/16$ fields) of that section (Figure 8.5). After clearing the side outcomes with an appropriate mask, we can add the partial products together to obtain the intended result. We say that section σ is *compressed into section* $\sigma + 1$.

In the same manner, section $\sigma+1$ would be compressed into section $\sigma+2$, section $\sigma-1$ into section σ, etc. Section $q-2$ is the last that may be non-zero and will be compressed into section $q-1$.

Now for the second complication: when multiplying our word with m_S^i, each of its bits will multiply all sections, not only the intended one. We wish to avoid any possibility that outcomes of multiplication with one section will clobber the compressed block of another. The solution lies in using two multipliers, G_1^i and G_2^i, such that G_1^i (resp. G_2^i) contains the bits of m_S^i related to sections in even (resp. odd) positions. So instead of one, we perform two multiplications, each one pertaining to one half of the sections; after each



The striped block is the result of section σ multiplied by bits intended for section $j > \sigma$.

Figure 8.6

multiplication we clear side outcomes with an appropriate mask (called D_1, D_2 resp.). The final result is the sum of the two partial results.

We claim that in each of the two multiplications, the compressed blocks (which are the desired part of the product) are not corrupted by undesired parts. To see this, we fix our attention at section σ (Figure 8.5 may help), and check how a multiplier, intended for another section $j \neq \sigma$, will shift its contents. Consider first the case $j < \sigma$. The most significant bit of section j has to be shifted at least $\frac{7}{16}L_j$ positions in order to reach its intended place in the result. Therefore, all shifts for fields of section j will be of $\frac{7}{16}L_j$ bits or more. Since we treat even and odd sections separately, $j \leq \sigma - 2$, and $\frac{7}{16}L_j \geq \frac{7}{2}L_{\sigma+1}$. Thus all the bits of section σ will be moved by these shifts at least $\frac{7}{2}L_{\sigma+1}$ positions to the left. This however throws them out of the word.

Now let $j > \sigma$. The largest shift that may be applied to section j is that which moves the least significant bit of that section into the compressed block. Thus the shift amount is bounded by $\frac{3}{2}L_j$. Since $j \geq \sigma + 2$, $\frac{3}{2}L_j \leq \frac{3}{8}L_\sigma$; therefore, the fields of section σ are shifted at most $\frac{3}{8}N$ positions to the left by this multiplication, as shown in Figure 8.6. Note that such products are formed for all $j > \sigma$ and they fall into approximately the same positions. Therefore, carries may propagate behind point ① (see figure). However, the carries produced by adding together q numbers can occupy at most $\lceil \log q \rceil$ bits. They will not reach point ②, because $\frac{1}{16}L_\sigma$ is always larger than $\log q$ (Lemma 8).

To sum up, compression of a key u (up to this stage) requires three masks, C_S, D_1 and D_2, and 64 multipliers, G_1^i, G_2^i ($i = 0, \ldots, 31$). The procedure for effecting the compression is

$$
\begin{array}{l}
v \leftarrow u \wedge C_S \\
z \leftarrow 0 \\
\textbf{for } i = 0, \ldots, 31 \\
\qquad z \leftarrow z + (v \cdot G_1^i) \wedge D_1 \\
\qquad z \leftarrow z + (v \cdot G_2^i) \wedge D_2 \\
\textbf{end}
\end{array}
$$

After this procedure, z will include $\hat{u}_i$ inside section $i+1$ for all i, or in short but informal notation: $z = \hat{u}_{q-2}\hat{u}_{q-3}\dots\hat{u}_1$. Because u has been cleared with C_S first, z contains the distinguishing fields in the respective compressed blocks, and all the rest is zeros.

We now explain how the masks and multipliers are obtained.

Fredman and Willard show how to find the multiplier m_S in $O(B^4)$ operations, the only difference from our requirements being that their digits are always single bits. We also need to split m_S into the 32 parts described above.

We show how their algorithm is adapted to our situation, without repeating it explicitly; all we need to know about it is that it decides one by one on bits to set in the multiplier, each one intended for shifting a particular distinguishing digit (i.e., bit) into the compressed block.

Assuming for the moment that we are still dealing with bits (so a "section" consists of N bits), replacing m_S by the 32 pieces m_S^i is simple: if the algorithm decides to set bit j in the multiplier, we will set it in m_S^i where $i = \lfloor \frac{j}{N/32} \rfloor$. Now, if the corresponding distinguishing digit in our algorithm happens to be a field of section σ, all the difference is that instead of setting bit j in the appropriate variable, we have to set bit $f_\sigma j$. Now, $\mathrm{bin}(f_\sigma j) = 2^{f_\sigma j}$. We obtain 2^{f_σ} in constant time (Lemma 11). It remains to raise it to the jth power. This takes $O(\log j)$ time using a standard algorithm. Since j is bounded by $\frac{3}{2}N$ (the maximal distance a field may have to be moved), and at most B bits have to be set, the total time for constructing the multipliers is $O(B^4 + B\log N) = O(B^4)$.

Consider now the masks D_1 and D_2. These masks are formed by first creating a mask for each distinguishing field, and then summing those of even-numbered sections in D_1 and the rest in D_2. The field masks are obtained as the multiplier is being built. For each $i \in D(S)$, say the algorithm decides that field i is going to be shifted j positions, putting $b_i = 2^{f_\sigma j}$ into the appropriate multiplier. A mask for the relocated field is obtained by shifting $F(i)$ to the same distance, namely, it equals $b_i F(i)$.

8.9 Rank Computation for Compressed Keys

The basic technique that enables rank computation with compressed keys to be fast is *multiple comparison*, which we describe first in a general fashion. Specific details are given later.

Let $x_1, \dots, x_k$ and $y_1, \dots, y_k$ be two lists of numbers. Our goal is to obtain, for each $1 \le i \le k$, a pair of bits that indicates which of the three relations: $x_i < y_i$, $x_i = y_i$ or $x_i > y_i$ holds. Assuming that for each i, we know how many bits to allocate for representing x_i or y_i (say b_i bits), and that $\sum_{i=1}^{k}(b_i + 1) \le w$, we can obtain our result as follows. Let X be the word

$$1x_k \dots 1x_2 1x_1$$

(where each x_i is allocated b_i bits), and similarly let Y be

$$0y_k \dots 0y_2 0y_1$$

extra zeros around these "fields" are permitted, but their locations in X and Y must correspond. By computing $X - Y$, the 1 before x_i will turn to 0 if and only if $x_i < y_i$. By

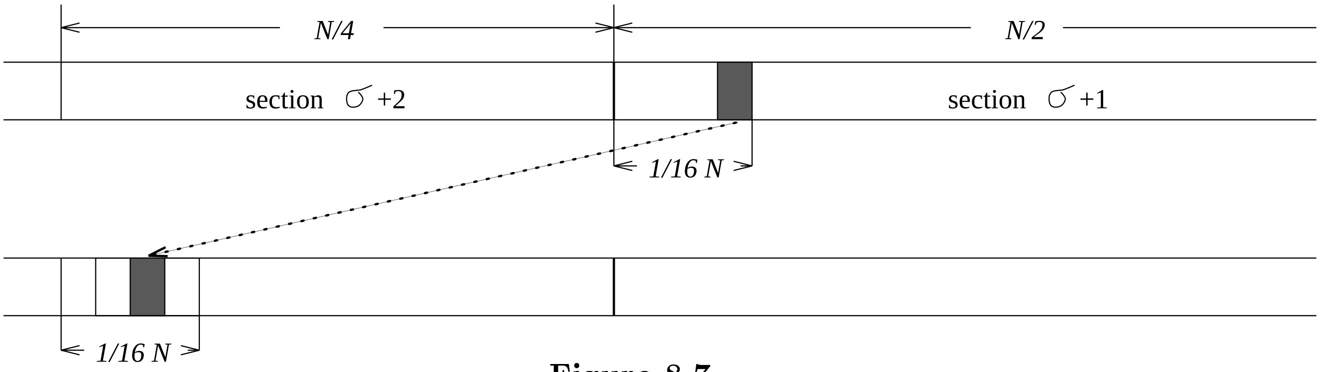


Figure 8.7

reversing the procedure (ones in Y and zeros in X), we obtain the flags for $x_i > y_i$. Together these flags indicate, for every i, which of the three relations holds. In our application, we want the result of the second comparison to be expressed in the inverse way, i.e., a flag of 1 if $x_i > y_i$. This is accomplished by Boolean negation.

Fusion is the technique of combining the set of compressed keys $\hat{S}$ into a single word which serves for fast rank computation using multiple comparison. Let $u^1, u^2, \ldots, u^B$ be the keys. Recall that for every key u, $\hat{u}$ has the form $\hat{u}_{\sigma_k}\hat{u}_{\sigma_{k-1}} \ldots \hat{u}_{\sigma_1}$ where $\hat{u}_{\sigma_j}$ is a compressed representation of section σ_j and $k < B$. We now view the compressed keys as numbers of up to B digits, where digit=compressed block. This observation should make it clear that in order to find the rank of $\hat{u}$ relative to $\hat{S}$, it suffices to compare each digit $\hat{u}_{\sigma_j}$ with all the corresponding digits $\hat{u}^i_{\sigma_j}$ (a total of B^2 comparisons). The outcome of the comparisons determines the rank uniquely. This suggests the following procedure:

(i) use multiple comparison to obtain quickly the results of the B^2 comparisons, expressed as B^2 pairs of bits.

(ii) use this $2B^2$-bit value to access a table where the corresponding rank is found. The table will be built in a preprocessing phase of the algorithm, and will be used for every node. So the $2B^2$ bits obtained from comparisons will have to be relocated into fixed positions, to fit a single precomputed table.

In the previous sections we described how when constructing a node, we obtain compressed representations of the B keys. As part of the node construction we will also fuse these compressed keys into a word X_S that allows for multiple comparison. Note that the identity of the sections participating in the compressed key is different from node to node, but is the same for all compressed keys of a certain node: it is the set $\bar{B}(S)$. As in the compression phase, we apply the next step in a parallel way to all the sections; the step shifts the compressed block into a certain position in the succeeding section.

We concentrate on the representative section σ. The compressed block $\hat{u}^j_\sigma$ is B^4 fields long and appears, for all j, in the place indicated in Figure 8.5, within section $\sigma + 1$. In order to fuse all these blocks together, we move them into consecutive niches of B^4 fields ($B^4 f_\sigma$ bits), which start in the last quarter of section $\sigma + 2$ (see Figure 8.7). Together the B blocks will occupy $B^5 = N/16$ fields, the upper quarter of section $\sigma + 2$. Thus the block from the first key, $\hat{u}^1_\sigma$, should be shifted $L_{\sigma+2}$ bits to the left; $\hat{u}^2_\sigma$ is shifted $L_{\sigma+2} + B^4 f_\sigma$ bits and so forth.

Recall that the last section that may be non-zero is section $q-2$; its compressed block lies within section $q-1$ and moves now into section q. Note also that the niches of $B^4 f_\sigma$ bits are contiguous; we do not dedicate an extra bit position for the flags as illustrated for the general technique. Our flag bits will occupy the most significant position of each niche, which goes to say that we really allow only $B^4 f_\sigma - 1$ bits to a compressed block. To ensure that this suffices, a little care is needed in the compression phase; we omit the details.

Relocating $\hat{u}^j_\sigma$ is done by multiplication with

$$b_{j,\sigma} = \mathrm{bin}(L_{\sigma+2} + (j-1)B^4 f_\sigma)$$

we will perform this for $j = 1, \ldots, k$ and add the results together. To take care of all the sections in parallel and without collisions, we repeat the method used in the compression phase. For each $\sigma \in \bar{B}(S)$, we compute $b_{j,\sigma}$ as we shortly explain, and put it into one of two multipliers: into H_1^j the multipliers for even sections, and into H_2^j the multipliers for odd sections. The proof that this way, side-outcomes of multiplications do not collide with intended outcomes is the same as for the compression phase. The special constant M_1 is used to clear side outcomes for the first multiplication, and M_2 clears side outcomes for the second one. We set

$$X_S = \sum_{j=1}^{k} \Big((H_1^j \cdot \hat{u}^j) \wedge M_1\Big) + \Big((H_2^j \cdot \hat{u}^j) \wedge M_2\Big) \ .$$

For computing $b_{j,\sigma}$, note that

$$b_{j,\sigma} = 2^{L_{\sigma+2}} + \left(2^{B^4 f_\sigma}\right)^{j-1} \ .$$

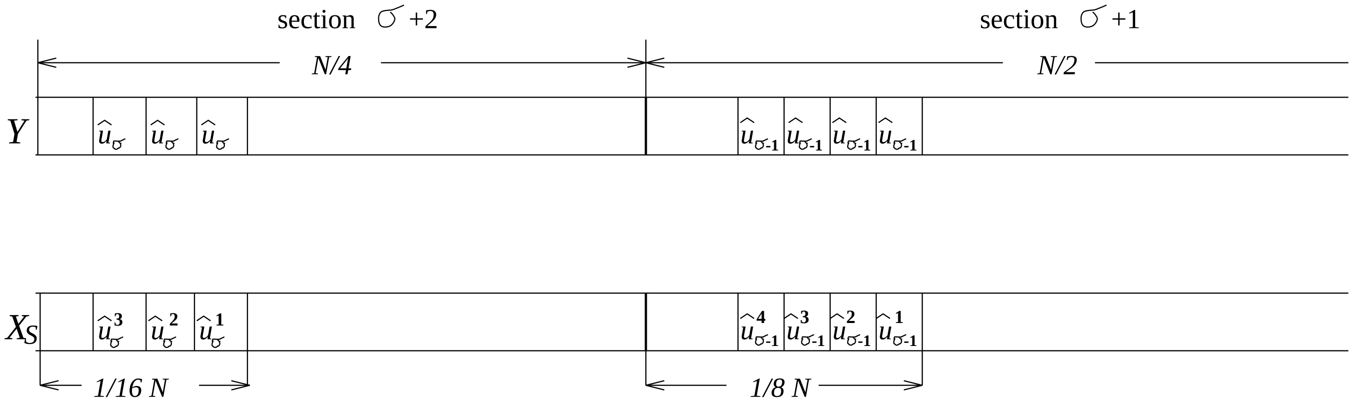
The values $2^{L_{\sigma+2}}$ and $2^{B^4 f_\sigma}$ are obtained in constant time (Lemma 11), and we loop on j with one multiplication per iteration, for a total of $O(B)$ time for each $\sigma \in \bar{B}(S)$, and $O(B^2)$ total time. We also compute (for $p = 1, 2$) $\mathcal{H}_p = \sum_{j=1}^{k} H_p^j$ and save it for use when searching the node as shown below.

When presented with a new key u, we compute $\hat{u}$ and then create a word Y where $\hat{u}_\sigma$ is repeated in all B niches of section $\sigma + 2$ (Figure 8.8). This allows for the multiple comparison operation. Y is computed by

$$((\mathcal{H}_1 \cdot \hat{u}) \wedge M_1) + ((\mathcal{H}_2 \cdot \hat{u}) \wedge M_2) \ .$$

We are now ready for multiple comparison using X_S and Y: to this end we use the mask M_5. Since the field size of section σ is four times $f_{\sigma+2}$, this mask has ones exactly in the high-order positions of the B^4-field niches. So our two sets of flags are obtained as

$$\mathcal{F}_1 = ((X_S \vee M_5) - Y) \wedge M_5$$



The upper word is obtained from the new key. The lower represents the set S of keys.

Figure 8.8

(the OR with M_5 sets the flag bits in X_S, and the AND clears everything but the flags);

$$\mathcal{F}_2 = \neg((Y \vee M_5) - X_S) \wedge M_5$$

(the OR with M_5 sets the flag bits in Y. The negation inverts the flags). The flag bits of $\mathcal{F}_1$ are in positions where a digit of u^j is smaller than the corresponding digit of u, and the flags of $\mathcal{F}_2$ are 1 where it is larger. Note that these flags number at most B^2; there are at most B distinguishing sections, in each of which we use at most B niches.

Having obtained these two words of flags, we will move the flags into fixed positions within the *buffer area* of the word. Recall that the buffer constitutes the last $s/4$ bits of the word. We use the rightmost B plots of the buffer area to receive the flags for the (at most B) distinguishing sections. More precisely, we will move the $2B$ flag bits for section σ_j into the jth plot (counting from the right end of the buffer). We put the B flag bits given by $\mathcal{F}_1$ in F/B bit intervals, in reverse order (that is, the flag corresponding to $\hat{u}^1_\sigma$ will be the leftmost in the plot, while in X_S, $\hat{u}^1_\sigma$ occupies the rightmost niche); see Figure 8.9. This reversal has no significance on the computation, since $\mathrm{rank}_{\hat{S}}(\hat{u})$ does not depend on the order in which $\hat{S}$ is given.

We put the flag bits from $\mathcal{F}_2$ adjacent to the flag bits from $\mathcal{F}_1$ (bit i from $\mathcal{F}_2$ next to bit i from $\mathcal{F}_1$). So we obtain, within plot j, B pairs of flag bits where pair i defines the relation between the jth digit of $\hat{u}$ and the corresponding digit of $\hat{u}^i$. This pair of bits (considered as a two-bit integer) serves as an indicator that has the value 1 if the digits are equal, 0 if $\hat{u}^i_{\sigma_j}$ is smaller, and 3 if $\hat{u}_{\sigma_j}$ is smaller. Here is an example:

$$\hat{u}^1 = 100 \quad \hat{u}^2 = 121 \quad \hat{u}^3 = 123 \quad \hat{u}^4 = 233 \quad \hat{u} = 211$$

$$\begin{array}{cccc} X_S & 1\,1\,1\,2 & 0\,2\,2\,3 & 0\,1\,3\,3 \\ Y & 2\,2\,2\,2 & 1\,1\,1\,1 & 1\,1\,1\,1 \\ \text{indicators} & 0\,0\,0\,1 & 0\,3\,3\,3 & 0\,1\,3\,3 \end{array}$$

After the relocation, we clear the rest of the word and obtain a word that contains the indicators only, which we denote by $I(\hat{u}, \hat{S})$. It will be used as an index to a pre-computed table that yields the rank of $\hat{u}$ relative to $\hat{S}$.

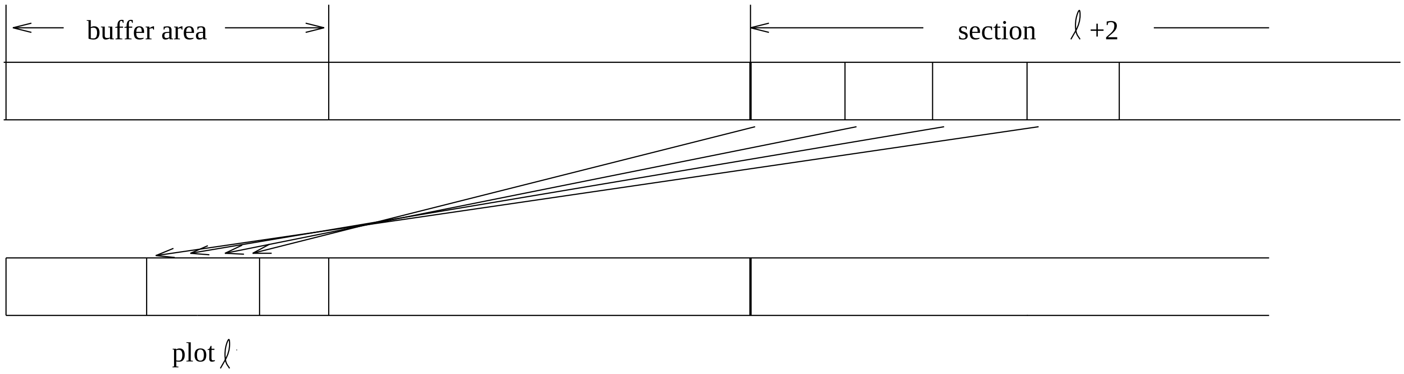


Figure 8.9

We now describe how the $2B^2$ flags can be relocated from their positions in $\mathcal{F}_p$ into the fixed positions in the buffer. Our goal is to compute a single multiplier $\mathcal{M}$ such that the desired relocation will be affected by simply multiplying $\mathcal{F}_1$ by $\mathcal{M}$ and $\mathcal{F}_2$ by $2\mathcal{M}$.

Lemma 13. *There is a multiplier $\mathcal{M}$ that moves the B flag bits from each of the distinguishing sections into their positions in the respective buffer plots. No other bits of the rightmost B plots of the buffer are set. This multiplier can be computed (from our special constants) in $O(B^2)$ time.*

Proof. $\mathcal{M}$ will be a sum of k multipliers, one for each distinguishing section. Let $\ell = \sigma_j$ where $1 \leq j \leq k$. Recall that the flags for section ℓ are the high-order bits of B blocks, each block of B^4 fields, that are located in the upper quarter of section $\ell + 2$. The multiplier M_ℓ has to move the flag bits from section $\ell + 2$ into the jth plot of the buffer, as shown in Figure 8.9.

Recall that the length of the buffer area is L_q, and that distance from the end of section $\ell + 2$ to the end of the word equals $L_{\ell+2}$. It may be verified with the help of the figure that the distance that flag-bit number t (counting from the left of the section) has to be moved equals

$$\begin{aligned} d_{\ell,t} &= (t-1)B^4 f_\ell + L_{\ell+2} - L_q + (\ell-1)F + \frac{F}{B}t \\ &= L_{\ell+2} - L_q + (t-1)(B^4 f_\ell + \frac{F}{B}) + (\ell-1)F + (\frac{F}{B}) \end{aligned}$$

We have

$$M_\ell = \mathrm{bin}(d_{\ell,1}, d_{\ell,2}, \ldots, d_{\ell,B})\,.$$

To see that only the intended bits of the rightmost buffer plot are set (and are not clobbered by carries), note that the difference between any two different $d_{\ell,t}$'s is greater than B^4F (F being the smallest field size). Thus, any multiplier bit not intended for a given flag will move it to a position at least $B^4 - 1$ plots away from the target plot, keeping it from clobbering the desired part either directly or through carries.

M_ℓ will be computed by summing the terms $2^{d_{\ell,t}}$ for $t = 1, \ldots, B$. Note that for $t > 1$,

$$d_{\ell,t} = d_{\ell,t-1} + B^4 f_\ell + \frac{F}{B}$$

so in each step of the summation we only have to multiply the previous term by $2^{B^4 f_\ell + F/B}$. The initial term is

$$2^{d_{\ell,1}} = 2^{L_{\ell+2} - L_q} \cdot 2^{(\ell-1)F} \cdot 2^{\frac{F}{B}} .$$

It follows that in order to complete the summation in $O(B)$ time it suffices to obtain, in constant time, the following values:

(i) $2^{B^4 f_\ell}$.

(ii) $2^{L_{\ell+2} - L_q}$.

(iii) $2^{(\ell-1)F}$.

(iv) $2^{\frac{F}{B}}$.

The first three of these values are considered in Lemma 11; The fourth is the precomputed constant K_1. □

Let us review the processes described for clarifying what is done at each stage — preprocessing, node construction and node search.

Preprocessing: a table is built that may be indexed with $I(\hat{u}, \hat{S})$ values and includes the corresponding rank. We give the construction of this table in the next subsection.

Node construction: ($O(B^4)$ time). We are given a set S of keys. The set of distinguishing positions, $D(S)$, is computed and so are the multipliers G_1^i, G_2^i and the masks C_S, D_1 and D_2 that are used to compress the keys. The fused word X_S is created, as well as the associated multipliers $\mathcal{H}_1$, $\mathcal{H}_2$ and $\mathcal{M}$. All these values are saved in the data structure.

Node search: (constant time). The new key u is compressed using the masks and multipliers saved in the node. The word Y is formed, multiple comparison is carried out and the value $I(\hat{u}, \hat{S})$ is obtained. Table lookup yields the desired rank.

8.10 The Lookup Table

The value $I(\hat{u}, \hat{S})$ contains B^2 digits (pairs of bits), $d_{B^2} d_{B^2-1} \cdots d_2 d_1$. The digit $d_{(j-1)B+i}$ is an indicator that signifies whether the jth digit of $\hat{u}^{B+1-i}$ (recall that the indicator for $\hat{u}^1$ is leftmost) is smaller, equal or larger than the jth digit of $\hat{u}$. Note that while the number of distinguishing sections is smaller than B, compressed keys are always represented in B niches, where the unused ones contain zeros; so it is correct to consider them as numbers of B digits.

The indicator bits are placed in particular positions within the buffer area of the word, as explained in the last subsection; for the moment we neglect this point and identify $I(\hat{u}, \hat{S})$ with the number

$$X = d_{B^2} d_{B^2-1} \cdots d_1$$

(we consider X as a ternary number, written using the set of digits $\{0, 1, 3\}$). We obtain X_i by selecting from X the indicators that pertain to $\hat{u}^{B+1-i}$:

$$X_i = d_{(B-1)B+i} \cdots d_{B+i} d_i .$$

Define Z to be the B-digit number $11 \cdots 1$.

Lemma 14. *If $I(\hat{u}, \hat{S}) = X$, then* $\mathrm{rank}_{\hat{S}}(\hat{u}) = \mathrm{rank}_{\{X_1,\ldots,X_B\}}(Z)$.

Proof. Let x, y be two B-digit numbers (in any given radix system). We define the *comparison word* $C(x, y) \in \{<, =, >\}^B$ as follows: the jth symbol of $C(x, y)$ gives the relation between the jth digits of x and y. For example:

$$\begin{array}{rcl} x & = & 6\,7\,4\,8\,1 \\ y & = & 5\,2\,4\,7\,1 \\ C(x,y) & = & >>=>= \end{array}$$

It is easy to see that the order relation between x and y is determined by $C(x, y)$ (specifically, they are equal if $C(x, y)$ is all $=$'s, otherwise the relation is given by the leftmost symbol different from $=$).

X_i and Z are both B-digit numbers over $\{0, 1, 3\}$. $\hat{u}^i$ and $\hat{u}$ are both B-digit numbers where a digit is a compressed block. For $1 \le j \le B$, digit j of X_i is an indicator that has the value 0,1 or 3 depending on the relation between the jth digit of $\hat{u}^i$ and the corresponding digit of $\hat{u}$:

— if the jth digit of $\hat{u}^i$ is smaller, the jth digit of X_i is 0, and is smaller than the jth digit of Z.

— if the jth digit of $\hat{u}^i$ is larger, the jth digit of X_i is 3, and is larger than the jth digit of Z.

— if the jth digits of $\hat{u}^i$ and $\hat{u}$ are equal, the jth digit of X_i is 1, and equals the jth digit of Z.

It follows that the jth symbol of $C(X_i, Z)$ equals that of $C(\hat{u}^i, \hat{u})$; thus $C(\hat{u}^i, \hat{u}) = C(X_i, Z)$, implying that $\hat{u}^i < \hat{u}$ if and only if $X_i < Z$. Therefore, $\mathrm{rank}_{\{\hat{u}^1,\ldots,\hat{u}^B\}}(\hat{u}) = \mathrm{rank}_{\{X_1,\ldots,X_B\}}(Z)$. □

The lemma shows that the $\mathrm{rank}_{\hat{S}}(\hat{u})$ can be correctly obtained from a lookup table that contains, for each possible value of $I(\hat{u}, \hat{S})$, the rank given by the lemma. To preset the table, we use a loop in which X ranges over the ternary numbers $00 \cdots 0$, ..., $33 \cdots 3$. For each value of X, we compute $\mathrm{rank}_{\{X_1,\ldots,X_B\}}(Z)$ and set it in the corresponding table entry.

As explained in the last subsection, each indicator (pair of bits) in $I(\hat{u}, \hat{S})$ is left-justified within a block of F/B bits. Altogether they occupy the rightmost B plots of the buffer area. In the implementation of this algorithm, X will be maintained such that each ternary digit is left-justified within a block of F/B bits. Hence X has the form of $I(\hat{u}, \hat{S})$ except for positioning within the word (X is right-justified in the word while $I(\hat{u}, \hat{S})$ is in the buffer area).

To perform operations on the digits of X, we use the following values.

For $p = 1, 2, \ldots, B^2$, a mask that selects the digit d_p:

$$S[p] = \underbrace{110 \cdots 0}_{pF/B}$$

1: $X \leftarrow 0$

2: $\mathit{Rank} \leftarrow 0$

3: for $i = 1, \ldots, B$ compute rank

4: $\quad \mathit{Rank} \leftarrow \mathit{Rank} + ((X \wedge T[i]) < L[i])$

5: $\quad\quad$ {comparison yields 1 for “true”, 0 for “false”}

6: $\mathit{Table}[X \cdot K_3] \leftarrow \mathit{Rank}$

7: $p \leftarrow 1$

8: while $p < B^2$ and $X \wedge S[p] = S[p]$ $\{d_p = 3\}$

9: $\quad p \leftarrow p + 1$

10: if $p = B^2$ stop

11: if $X \wedge S[p] = 0$ then

12: $\quad X \leftarrow X + A[p]$ $\{d_p \leftarrow 1\}$ advance counter

13: else

14: $\quad X \leftarrow X + 2A[p]$ $\{d_p \leftarrow 3\}$

15: endif

16: $X \leftarrow X \wedge \neg(A[p] - 1)$ {clear lower digits}

17: go to 2

Figure 8.10

A word that contains a 1-digit at position p ($p = 1, 2, \ldots, B^2$)

$$A[p] = \underbrace{010\cdots0}_{pF/B}$$

For $i = 1, 2, \ldots, B$, a mask that selects the B digits that comprise X_i:

$$T[i] = \underbrace{110\cdots0}_{F}\ \underbrace{110\cdots0}_{F}\ \cdots\ \underbrace{110\cdots0}_{F}\ \underbrace{110\cdots0}_{iF/B}$$

In computing $\mathrm{rank}_{\{X_i\}}(Z)$, we have to compare Z to each X_i. To simplify this comparison, we use the following values (for $i = 1, 2, \ldots, B$):

$$L[i] = \underbrace{010\cdots0}_{F}\ \underbrace{010\cdots0}_{F}\ \cdots\ \underbrace{010\cdots0}_{F}\ \underbrace{010\cdots0}_{iF/B}$$

$L[i]$ contains the digits of Z $(11\cdots1)$ in the positions corresponding to X_i. All these masks are obtained in $O(B^2)$ time given the constant K_1.

The algorithm is given in detail in Figure 8.10. Note that the position within the word is rectified in the line that sets the table (Line 6) by shifting the value of X into the buffer area.

The running time of this algorithm can be easily bounded by

$$O(3^{B^2} B^2) = O(3^{(\log n)^{1/3}} (\log n)^{1/3}) = o(n)\,.$$

8.11 The Weakly Non-Uniform Algorithm

If we allow a weak non-uniformity, the range of word lengths where Modified Fusion Trees allow fast sorting becomes larger.

Theorem 9. *There is a weakly non-uniform algorithm, based on Modified Fusion Trees, that sorts in $o(n \log n)$ time provided* $\log\log\log w = o(\log n)$.

Note that for unrestricted w, the solution given in Theorem 4 uses strong uniformity. Thus there is still a gap between the conditions that allow fast sorting using weak versus strong non-uniformity.

Proof. We assume $\log\log\log w$ to be bounded by some function $f(n)$ which is $o(\log n)$. We define $s = 2^{f(n)+\log\log n}$. The constants s and 2^{w-s} are now assumed to be hard-coded and allow the Kirkpatrick-Reisch algorithm to be run on the upper s bits of the word. It will run in $O(n(f(n) + \log\log n)) = o(n \log n)$ time.

We define q, B, N and F as before. The bound on w was only used in the analysis of the MFT algorithm in two ways. The first involves the time for constructing special constants. Since our list of special constants is of fixed length, weak non-uniformity allows us to ignore the time required for computing them. Secondly, it was used in proving Lemma 8. Direct calculation of the expressions involved shows that the lemma still holds. However, the observation made in the course of the proof given, stating that $\log qN = O(\log n)$, is not correct now. This observation is used in the MSD procedure, since the position of a field is a number of $\log qN$ bits. When $\log qN \le \log n$, we can move this number from the buffer area using SHIFT. To handle the situation when $\log qN$ exceeds $\log n$, we break the number into parts of $\log n$ bits each, and use SHIFT to move them one by one to the desired positions. This operation dominates the time of the MSD procedure, which becomes

$$O(\frac{\log qN}{\log n}) = O(\frac{\log\log w}{\log n}) = O(\frac{2^{f(n)}}{\log n}) .$$

Observe that only during node construction do we actually have to obtain the MSD position as a separate integer number; in node search, we need the relation between two MSD positions, which may be obtained by comparing the words that contain the positions in their buffer areas. Therefore the node search time remains $O(1)$. The time for node construction, however, changes to

$$O(B\frac{2^{f(n)}}{\log n}) = o(2^{f(n)}) .$$

The increase in node construction time necessitates a change in the data structure. This change has already been discussed by Fredman and Willard. It involves removing some of the lowest levels of the B-tree, replacing the subtrees of these levels by balanced binary search trees. Our structure becomes a B-tree whose leaves are identified with roots of binary trees of size bounded by $2^{f(n)}$. This reduces the number of updates to B-tree nodes from n to $n2^{-f(n)}$, which makes the total time for updating the B-tree bounded by $n2^{-f(n)} \cdot o(2^{f(n)}) = o(n)$, and incurs an addition of $O(f(n))$ time to each data-structure operation. This totals in $O(nf(n)) = o(n \log n)$; so the time to sort remains $o(n \log n)$. □

9. Concluding Remarks

9.1 Non-Uniform in what?

We defined a uniform program as one that is free from dependence on both n and w. Denote by $\mathcal{NU}(n)$ the class of programs which are non-uniform only with respect to n. For such a program, let $L_0(n)$ be the length (in significant bits) of the largest program constant dependent on n. Theorem 7 may be extended to show that $\mathcal{NU}(n)$ programs are not stronger than uniform programs except that the range of w, where fast sorting is possible, may grow:

Theorem 10. *An $\mathcal{NU}(n)$ RAM2 program requires $\Omega(n \log n)$ time to sort n numbers given $w \geq \dot{\Omega}(2^{n \log n}) + L_0(n)^4$, and $a \leq w - w^{\varepsilon}$ for some $\varepsilon > 3/4$.*

There is no essential change in the proof; the modified bound on w ensures that the basis of the induction in Lemma 5 will hold.

9.2 Optimality of Upper Bounds

For Theorem 1, our upper bound is given by the Fusion Tree algorithm. The precise running time of this algorithm depends on the word length: it is $\Theta(n\sqrt{\log n} + n\frac{\log n}{\log w})$. This result is not known to be optimal within the model, so establishing a tight bound remains an important open problem.

The upper bound of Theorem 2 is the same for words short enough ($w = \dot{o}(2^{n\sqrt{\log n}})$), because the running time of the Fusion Tree algorithm dominates the $O(\log w)$ cost of preprocessing. Hence in this case, the result is not known to be optimal. For longer words, the preprocessing time dominates, setting the upper bound at $O(\log w)$. This is optimal by Theorem 7.

Theorem 3 uses the Fusion Tree algorithm when $a = w$, so in this case, the running time is $\Theta(n\sqrt{\log n} + n\frac{\log n}{\log w})$. Instead of Fusion Trees, we could use Kirkpatrick-Reisch, and obtain an upper bound of $O(n \log w)$, which is better whenever $\log w < \sqrt{\log n}$. In either case we don't know if the result is optimal. When $a < w$, additional $\Theta(n(w-a))$ time is required for the radix-sort phase. This is optimal by Lemma 2 (choose $c = \frac{w-a}{\log n}$).

Theorem 4, uses Modified Fusion Trees when $a = w$, and w is large; the running time of this algorithm is $\Theta(n \log n / \log \log n)$, so it will dominate the $O(\log w)$ term (introduced by the stage of constructing constants) whenever $\log w < n \log n / \log \log n$. For this range, it is not known to be optimal. When w is larger, we obtain the optimal time of $O(\log w)$. When $a < w$, additional $\Theta(n(w-a))$ time is required for the radix-sort phase, which is optimal.

9.3 Other Open Problems

As pointed out in the introduction, one major open problem is to close the gap between the memory restriction in Theorem 7 and our conjecture that this restriction is redundant.

Another question arises by comparing Theorems 3 and 6: Is strong non-uniformity required for fast sorting without $\mathcal{DP}$ and $\mathcal{SW}$? As shown in the last chapter, strong non-uniformity is not required if $\log\log\log w = o(\log n)$. Our conjecture, yet unproved, is that it becomes essential when words are large enough.

Finally, we may ask how adding stronger features (e.g., shift) will affect the running time.

9.2 Related Work

The Kirkpatrick-Reisch algorithm has been rediscovered by Matias and Vishkin [11]. The Paul-Simon lower bound has been reproved by Bshouty [7], who also treats division over a field containing the integers.

Appendix A. Procedures For Section 2

procedure B
$B \leftarrow 1$
$C \leftarrow 1 \qquad \{C = B^6\}$
$D \leftarrow 1 \qquad \{D = 2^{(\log B)^2}\}$
while $64C < w$ and $2B^2D < n$
    $C \leftarrow 64C$
    $D \leftarrow 2B^2D$
    $B \leftarrow 2B$
end while

procedure G
$C \leftarrow 1$
$D \leftarrow 2^F \qquad \{D = 2^{CF}\}$
$G \leftarrow 2^{F-1} \qquad \{G = \underbrace{2^{F-1} \ldots 2^{F-1}}_{C}\}$
while $C < B$
    $C \leftarrow 2C$
    $G \leftarrow DG + G$
    $D \leftarrow D^2$
end while

Appendix B. Procedures for Section 5

The following procedures implement double-precision multiplication of numbers in $N(l, p, q)$, i.e., numbers of the form

$$\begin{array}{cccc} q & p & l & 0 \\ \hline \sigma\sigma\sigma\sigma\sigma & /// & \tau\tau\tau\tau\tau & /// \end{array}$$

in such a way that when the multiplication primitive is actually applied, the numbers involved will belong to $N^*(l', p', q')$ where $(l', p', q') = (2l, 2p - w, q + l)$. A convenient notation for this part will be the following: $N(l, p, q, \sigma[, \tau])$ is the set of numbers shown above with specific values of σ [and τ]. Thus $N^*(l, p, q) = N(l, p, q, 0, 0)$.

We give the procedures in the standard algol-like dialect. We denote the bitwise complement of A by $\overline{A}$. The operator $\times$ stands for double-precision multiplication, yielding a result in the form (H, L). Similarly, where the operators $+$ and $-$ give a result as a pair of variables, they should be interpreted as double-precision operators. Two procedures are defined: MUL multiplies X by Y and returns the result in H, L. MULA is an auxiliary procedure. The procedures operate as follows: MUL reduces multiplication of numbers in $N(l, p, q)$ to multiplications of numbers in $N(l, p, q', 0)$. For each such multiplication, it calls MULA, which further reduces the operation to multiplying members of $N(l', p', q', 0, 0)$.

Explanation of MUL. Let X an Y be the input numbers. MUL first computes A and B by masking out the σ bits of X and Y, repsectively, so $A, B \in N(l, p, q, 0)$. It proceeds to exploit the following observations:

(i) if $X = A$ and $Y = B$, than $XY = AB$.

(ii) if $X \neq A$ and $Y = B$, than $X = A + 2^w - 2^q$ and $XY = AB + 2^w B - 2^q B$.

(iii) if $X = A$ and $Y \neq B$, than $Y = B + 2^w - 2^q$ and $XY = AB + 2^w A - 2^q A$.

(iv) if $X \neq A$ and $Y \neq B$, than $X = A + 2^w - 2^q$, $Y = B + 2^w - 2^q$ and
$XY = AB + (2^w B - 2^q B) + (2^w A - 2^q A) + (2^{2w} - 2^{w+q+1} + 2^{2q})$.

Note that $2q > w$; so all the terms in the last parentheses affect only the high-order word of the product. This fact is used in the last assignment, where the high-order part of this constant is added directly to H, using ordinary addition (carry from the high-order word does not affect the result).

Explanation of MULA. Let X an Y be the input numbers, both in $N(l, p, q', 0)$. MULA first computes A and B by masking out the τ bits of X and Y, repsectively, so $A, B \in N(l, p, q', 0, 0)$. It proceeds to exploit the following observations:

(i) if $X = A$ and $Y = B$, than $XY = AB$.

(ii) if $X \neq A$ and $Y = B$, than $X = A + 2^p - 2^l$ and $XY = AB + 2^p B - 2^l B$.

(iii) if $X = A$ and $Y \neq B$, than $Y = B + 2^p - 2^l$ and $XY = AB + 2^p A - 2^l A$.

(iv) if $X \neq A$ and $Y \neq B$, than $X = A + 2^p - 2^l$, $Y = B + 2^p - 2^l$ and
$XY = AB + (2^p B - 2^l B) + (2^p A - 2^l A) + (2^{2p} - 2^{p+l+1} + 2^{2l})$.

Note that $2p > w$ while $2l < p + l + 1 < w$. Hence the sum in the last parentheses breaks into two parts: $-2^{p+l+1} + 2^{2l}$ has to be added to the low-order word, and 2^{2p} to the high-order word. However, the former is a negative number, while a machine word is always interpreted as non-negative; so, to add this value, we subtract its complement $D = 2^{p+l+1} - 2^{2l}$. In order to perform both operations at once (using a double-precision subtraction), we complement the high-order part as well.

procedure MUL(in X, Y; **out** H, L**)**
    $M \leftarrow (-2^q)$
    $A \leftarrow X \wedge (\overline{M})$, $B \leftarrow Y \wedge (\overline{M})$
    MULA(A, B, H, L)
    if $X \neq A$ **then**
        $H \leftarrow H + B$
        MULA$(2^q, B, C, D)$
        $(H, L) \leftarrow (H, L) - (C, D)$
    endif
    if $Y \neq B$ **then**
        $H \leftarrow H + A$
        MULA$(2^q, A, C, D)$
        $(H, L) \leftarrow (H, L) - (C, D)$
    endif
    if $X \neq A$ **and** $Y \neq B$ **then**
        $H \leftarrow H - 2^{q+1} + 2^{2q-w}$
    endif
end

procedure MULA(in X, Y; **out** H, L**)**

$M \leftarrow 2^p - 2^l$

$A \leftarrow X \wedge (\overline{M}), \quad B \leftarrow Y \wedge (\overline{M})$

$(H, L) \leftarrow A \times B$

if $X \neq A$ **then**

$(C, D) \leftarrow 2^p \times B$

$(E, F) \leftarrow 2^l \times B$

$(H, L) \leftarrow (H, L) + (C, D) - (E, F)$

endif

if $Y \neq B$ **then**

$(C, D) \leftarrow 2^p \times A$

$(E, F) \leftarrow 2^l \times A$

$(H, L) \leftarrow (H, L) + (C, D) - (E, F)$

endif

if $X \neq A$ **and** $Y \neq B$ **then**

$(C, D) \leftarrow (-2^{2p-w}, 2^{p+l+1} - 2^{2l})$

$(H, L) \leftarrow (H, L) - (C, D)$

endif

end

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass. 1974.

[2] M. Ajtai, M. Fredman and J. Komlos, "Hash functions for priority queues," *Information and Computation* 63 (1984) 217–225.

[3] R. Bayer and E.McCreight, "Organization and maintenance of large ordered indices," *Acta Informatica* 1 (1972), 173–189.

[4] A. M. Ben-Amram, "Some notions on notations," *SIGACT News* 22:4 (1991) 60–62.

[5] A. M. Ben-Amram and Z. Galil, "On pointers versus addresses," *J. of the ACM* 39:3 (1992) 617–648.

[6] N. H. Bshouty, "Lower bounds for the complexity of functions in a realistic RAM model," *Proc. Israel Symposium on the Theory of Computing and Systems*, Haifa, 1992.

[7] N. H. Bshouty, "Lower bounds for the complexity of functions in random access machines," *J. of the ACM* 40:2 (1993) 211–223.

[8] M. L. Fredman and D. E. Willard, "Surpassing the information theoretic barrier with Fusion Trees," *J. of Computing and System Sciences* 47:3 (1993), 424–436.

[9] D. Kirkpatrick and S. Reisch, "Upper bounds for sorting integers on random access machines," *Theoretical Computer Science* 28:3 (1984) 263–276.

[10] D. E. Knuth, *The Art of Computer Programming,* vol. 3: sorting and searching. Addison-Wesley, Reading, Mass. 1973.

[11] Y. Matias and U. Vishkin, "On parallel hashing and integer sorting", *J. of Algorithms* 12:4 (1991) 573–606.

[12] W. Paul and J. Simon, "Decision trees and random access machines," in *Logic and Algorithmic*, monographie no30 de l'enseignement mathématique, Université de Genève 1982.