DIMACS Technical Report 96-14
July 1996

Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations

by

Mikkel Thorup[1]
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
mthorup@diku.dk, http://www.diku.dk/~mthorup

[1]Part of this research was done while the author visited DIMACS.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

A randomized sorting algorithm is presented, doing as described in the title.

1 Introduction

In this paper we consider sorting on a very simple RAM where the only word-operations are addition, shift, and bit-wise boolean operations. Besides these word-operations, we have direct and indirect addressing, jumps, and conditional statements. Such a RAM has been referred to as a *Practical RAM* [Mil96]. In this paper we show

Theorem 1 *On a Practical RAM, there is a randomized algorithm sorting n words in $O(n \log\log n)$ time and linear space.*

The above algorithm only makes shifts by powers of two, and it only needs $O(\log n)$ random words. Our time bound matches that of the current fastest sorting algorithm by Andersson, Hagerup, Raman, and Nilsson [AHNR95]. Their algorithm has two variants: one is deterministic uses space $2^{\varepsilon w}$, where w is the word length and ε is a positive constant. Thus the space is unbounded in terms of n. The other variant is randomized and uses linear space like ours, but it uses multiplication.

Multiplication is considered expensive from several perspectives. From a theoretical view-point, it is known that the minimal depth of a polynomial sized circuit implementing multiplication is $\Theta(\log w / \log\log w)$ [BH89], which is unbounded in n. In contrast, all the Practical RAM operations are in AC^0, that is, they are all implementable by polynomial sized constant depth circuits.

From a practical view-point, we may circumvent the theoretical lower-bound by pipelining several multiplications at the time. Speed may be gained by tabulating the multiplication of small numbers, and by having many adders working in parallel. Nevertheless, multiplication takes longer time than the AC^0 operations of the Practical RAM.

A more serious objection to multiplication is that the cost of implementing a fast multiplication grows rapidly with the word-size, e.g. for addition based multiplication, we could use $\Theta(w)$ adders working in parallel.

It is a trend in modern processor design to increase the word-length and the band-width of the bus. For example, on a modern alpha-processor, the word length is 64 bits while the band-width is 128. This development increases the potential for exploiting processor parallelism in sequential computing. Restricting ourselves to Practical RAMs with no multiplication, makes it easier to increase the word length. Indeed there are RISC architectures with no direct support for multiplication. Theorem 1 states that the restriction to Practical RAMs does not have any asymptotic influence on the running time of sorting. Hence Practical RAMs may be used for future sorting co-processors with large word length.

The problem of doing $o(n \log n)$ sorting with AC^0 operations was posed by Fredman and Willard [FW93] when they presented the first $o(n \log n)$ sorting algorithms. Both their deterministic $O(n \log n / \log\log n)$ algorithm and their randomized $O(\sqrt{\log n})$ algorithm were heavily based on multiplication. The above mentioned deterministic algorithm of Andersson *et.al.* solves the problem, but uses space unbounded in terms of n. Since then Raman has found a deterministic Practical RAM algorithm running in time $O(n\sqrt{\log n \log\log n})$ and linear space. Also, in [AMRT96] is presented a randomized linear space algorithm with running time $O(n(\log\log n)^2)$. However, that algorithm is based on some powerful non-standard AC^0operations. As stated in Theorem 1, in this paper we reduce the running time to $O(n \log\log n)$ and use standard operations only.

Techniques

Previous randomized linear space $O(n \log\log n)$ [AHNR95] makes extensive use of multiplication based hashing. More precisely, they use hashing to identify duplicates in a list of keys.

Lemma 2 *On a Practical RAM, we can sort and time $O(n + \sum_{i=0}^{\log\log n} D(n, w/2^i))$ where $D(n, b)$ is the time it takes to identify duplicates in a list of n b-bit keys. The space consumption is that of identifying the doublicates.*

If we use universal hashing [CW79] as in [AHNR95], the identification of duplicates is done in linear time on the unit cost RAM: simply scan the keys one by one, adding each key to the list of previous identical keys, if any. However, if follows from [AMRT96], that if we restrict ourselves to AC^0-operations, any such "on-line" approach for finding duplicates, is going to take time $\Omega(n\sqrt{\log n/\log\log n})$. In fact, the lower-bound holds more generally for any instruction set if we say that the cost of an operation is the minimal depth of a polynomial sized circuit computing it. On the other hand, in [AMRT96] it is shown that the real "off-line" duplicates problem can be solved in time $O(n \log\log n)$ using some power-full non-standard AC^0-operations. Using Lemma 2, this leads to an $O(n(\log\log n)^2)$ AC^0 sorting algorithm. In this paper, we show

Theorem 3 *We can identify the duplicates in a list of n (w/q)-bit keys in $O(n+n\log\log n/q)$ expected time by a randomized Practical RAM algorithm using linear space.*

Together with Lemma 2, this immediately implies Theorem 1.

In [Tho96] is given a general reduction from amortized monotone priority queues to sorting. Here a *monotone* priority queue is a priority queue where the minimum is non-decreasing. This restriction does not affect most gready algorithms. Thus, we have the following consequence of Theorem 1:

Corollary 4 *There is a monotone Practical RAM priority queue supporting* `insert`*,* `delete`*, and* `extract-min` *in $O(\log\log n)$ amortized time, where n is the maximal number of keys in the queue.*

2 Preliminaries

The paper is primarily concerned with manipulation of bit-strings. We use multiplication notation for concatenation of strings. In order to avoid ambiguity, we use $\underline{0}$ and $\underline{1}$ for bits that are part of bit-strings. Thus $\underline{1} \cdot \underline{0} = \underline{10}$ while $1 \cdot 0 = 0$. Consider a bit string $x = b_0 \cdots b_{l-1}$, $b_i \in \{\underline{0}, \underline{1}\}$. Then $x[i]$ denotes b_{i-1}, $x[i..j]$ denotes $x[i] \cdots x[j]$, and $|x| = l$. Moreover, the number of $\underline{1}$s in a string x is denoted $\|x\|_1$. If $\|x\|_1 = 0$, we say that x is *empty*. For bit-strings x and y of equal length, we call $\|x \oplus y\|_1$ the *Hamming distance* from x to y. The set $\{y | \|x \oplus y\|_1 \leq r\}$ is called the *Hamming ball* with radius r and center x.

Of more non-standard notation, we use $x\langle i\rangle_k$ to denote $x[ik..(i+1)k-1]$. Also, for q dividing $|x|$, $x\langle i\rangle_{/q} = x\langle i\rangle_{|x|/q}$. Thus, the $x\langle i\rangle_k$ notation is used for dividing x into pieces of length k while the $x\langle i\rangle_{/q}$ notation is used to divide x into q equal-sized pieces. For example, $x\langle 0\rangle_{/2}$ and $x\langle 1\rangle_{/2}$ are the first and the second half of x.

We use $\mathtt{int}(x)$ to denote the number represented by x, i.e. $\mathtt{int}(x) = \sum_{i=1}^{|x|} 2^{|x|-i}\mathtt{int}(b_i)$, where $\mathtt{int}(\underline{0}) = 0$ and $\mathtt{int}(\underline{1}) = 1$. We use $\mathtt{str}(i,l)$ to denote the string x of length l such $\mathtt{int}(x) = i \bmod 2^l$. Note that if l is not long enough to represent i, we are discarding the most significant bits. For two bit-strings x and y, $x+y$ denotes $\mathtt{str}(\mathtt{int}(x)+\mathtt{int}(y),|x|)$.

The operators $\wedge$, $\vee$, and $\oplus$ to denote 'and', 'or', and 'exclusive-or'. If $\circ$ is any of these bit operations and x and y are bit-strings with $|x| \le |y|$, then $x \circ y$ denotes $\prod_{i=0}^{|x|-1}(x[i] \circ y[i]) = (x[0] \circ y[0]) \cdots (x[|x|-1] \circ y[|x|-1])$. Note the implicit truncation of y to the length of x. For example, we can now define negation by $\neg x \equiv x \oplus \underline{1}^\infty$. Then for minus, $-x = \neg x + \underline{1}$. Generally, given a bit string x and a number $l > |x|$, we assume that $x^\infty[0..l]$ is computed by $\log l - \log |x|$ squarings starting from x, i.e. $x^\infty[0..l] = x^{2^{\log l - \log |x|}}$. Here, and everywhere else in the paper, log is understood to be base 2. Also, most numbers are implicitly assumed to be powers of two, so that we avoid rounding problems. Finally, we have the shift operations: $x \uparrow d$ denoting the string y of length $|x|$ such that $y[(i+d) \bmod |x|] = x[i]$ for $i = 0, \ldots, |x|-1$. We use $x \downarrow d$ as a shorthand for $x \uparrow (-d \bmod |x|)$. Note that our shifts are cyclic.

As an exercise in our notation, we implement a standard routine that we will use frequently. Similar implementations may be found in, for example, [AH92].

Algorithm A: $\mathtt{Mask}(x,k)$. Returns y where $y\langle i\rangle_k$ equals $\underline{1}^k$ if $x\langle i\rangle_k \ne \underline{0}^k$; $\underline{0}^k$ otherwise.

A.1. $x_1 = (x \wedge (\underline{0}\,\underline{1}^{k-1})^{|x|/k}) + (\underline{0}\,\underline{1}^{k-1})^{|x|/k}$.

A.2. $x_2 = (x \vee x_1) \wedge (\underline{1}\,\underline{0}^{k-1})^{|x|/k}$ — $x_2\langle i\rangle_k$ equals $\underline{1}\,\underline{0}^{k-1}$ if $x\langle i\rangle_k \ne \underline{0}^k$; $\underline{0}^k$ otherwise.

A.3. $x_3 = x_2 \uparrow k-1$.

A.4. $x_4 = x_3 + (\underline{0}\,\underline{1}^{k-1})^{|x|/k}$.

A.5. Return $x_4 \oplus (\underline{0}\,\underline{1}^{k-1})^{|x|/k}$.

Assuming that the relevant prefixes of $(\underline{0}\,\underline{1}^{k-1})^{|x|/k}$ and $(\underline{1}\,\underline{0}^{k-1})^{|x|/k}$ are pre-computed, the algorithm only makes a constant number of string operations. If x fits in one word, $\mathtt{Mask}(x,k)$ is computed in constant time. Otherwise, suppose that $|x| > w$ but $k \le w$ where w is the word-length. Assuming that x is distributed over $q = |x|/w$ words $x_1, \ldots, x_q$, we simply compute $\mathtt{Mask}(x,k)$ as $\mathtt{Mask}(x_1,k) \cdots \mathtt{Mask}(x_q,k)$. Here, we are exploiting that the carry in step A.4 does not propagate between pieces. Conversely, suppose that we are considering several keys $x_1, \ldots, x_q$, $|x_i| = w/q$, packed in one word, then $\mathtt{Mask}(x_1,k) \cdots \mathtt{Mask}(x_q,k)$ may be computed as $\mathtt{Mask}(x_1 \cdots x_q, k)$. Thus, provided packing of keys in words, the amortized cost of computing $\mathtt{Mask}(x,k)$ for a key x and $k \le w$ is $O(|x|/w)$, which may be less than a constant. The above type of reasoning is straightforward, and in the rest of the paper we will often be less explicit about it.

3 The general approach

We will borrow the general approach from [AMRT96]. We are starting with a sequence X of n keys, each taking up one word. Our aim is to shorten the keys, so that they can be sorted in linear time (sorting implies that we group duplicates); either by packed sorting [AH92, AHNR95], or by radix sort with character of length $\log n$, depending on the relationship between n and w.

First, we have a *clustering phase* where we compute a short *signature* of each key so that with high probability, if to keys get the same signature, they have small Hamming distance. Using sorting on the short signatures, we cluster keys with the same signatures into Hamming balls of small radius.

Next, we have a *cluster bustering phase*. For each cluster Y of keys with the same signature, we select an arbitrary key $y_0 \in Y$, and compute the set $Y \oplus y_0 = \{y \oplus y_0 | y \in Y\}$. If r is the maximal difference between any two keys in Y, then no key in $Y \oplus y_0$ contains more than r $\underline{1}$s, i.e. $\forall y \in Y : \|y \oplus y_0\|_1 \leq r$. We then need to develop methods reducing keys with few $\underline{1}$s injectively to keys that are short enough for linear time sorting.

All the sets $Y \oplus y_0$ may be busted simultaneously. The shortest keys we will consider are of lenght $w/(\log n)^{\Theta(1)}$, so, essentially, we can always assume that we have enough keys to fill the words.

4 Long words

In this section, we prove Theorem 3 under the following assumption:

$$\log w = \omega(\log \log n) \iff w = (\log n)^{\omega(1)} \tag{1}$$

As pointed out in [AHNR95], the results from [AH92] imply

Lemma 5 *On an Practical we can sort n keys of length $O(w/(\log n \log \log n))$ in time $O(n)$ and space $O(n)$.*

Hence, in this section, we goal is to reduce the length of our keys to $O(w/(\log n \log \log n))$. Thus, assume that our keys have lenght $b = \omega(w/(\log n \log \log n))$. Set $k = w/(\log n \log \log n)$, and $q = b/k$. Let α be a random string of b bits (each bit is independently set to $\underline{1}$ with probability 1/2). A signature of length k is computed as follows.

Algorithm B: $\texttt{Sign}_\alpha(x)$

B.1. $y = x \wedge \alpha$.

B.2. Return $\bigoplus_{i=1}^{q} y\langle i \rangle_{/q}$.

The following lemma states that our signature separates keys that differ in many bit-positions.

Lemma 6 *For any two words x and y, if α is randomly chosen and $\|x \oplus y\|_1 = r$, then* $\Pr(\texttt{Sign}_\alpha(x) = \texttt{Sign}_\alpha(y)) \leq 2^{-r/q}$.

Proof: Let $z = x \oplus y$. Then $\texttt{Sign}_\alpha(z) = \texttt{Sign}_\alpha(x) \oplus \texttt{Sign}_\alpha(y)$. We are searching the probability that $\texttt{Sign}_\alpha(z) = \underline{0}^k$.

For $j = 1, \ldots, k$, set $I(j) = \{i | z\langle i \rangle_k[j] = 1\}$. Then $\texttt{Sign}_\alpha(z)[j] = \bigoplus_{i \in I(j)} \alpha\langle i \rangle_k[j]$. Since each $\alpha\langle i \rangle_k[j]$ is independently chosen to be $\underline{1}$ with probability 1/2, it follows that $\Pr(\texttt{Sign}_\alpha(z)[j] = \underline{1}) = 1/2$ if $I(j) \neq \emptyset$; 0 otherwise. Since $\|z\|_1 = r$ and $|I(j)| \leq q$ for all j, it follows that $I(j) \neq \emptyset$ for at lest r/q values of j. ∎

We will now show that in fact, we can compute all the signatures in linear time. Thereby we will reveal some of the basic techniques used in this paper. First, define

$$\begin{aligned}\texttt{Spread}(x,k) &\equiv (x \oplus (\underline{1}^k\underline{0}^k)^\infty)\,((x \oplus (\underline{0}^k\underline{1}^k)^\infty) \downarrow k)\\ \texttt{Collect}(x,k) &\equiv x\langle 0\rangle_{/2} \vee (x\langle 1\rangle_{/2} \uparrow k).\end{aligned}$$

Then $x = \texttt{Collect}(\texttt{Spread}(x,k),k)$.

Algorithm C: Let $\vec{x} = x_1 \cdots x_n$ be our list keys, each of length b. Let $k = b/q$. The algorithm computes the list $\texttt{Sign}_\alpha(x_1)\cdots\texttt{Sign}_\alpha(x_n)$.

C.1. $\vec{y} := \vec{x} \wedge \alpha^\infty$

C.2. Repeat $\log q$ times:

C.2.1. $b := b/2$;

C.2.2. $\vec{x} := (\vec{x} \wedge (\underline{1}^b\underline{0}^b)^\infty) \oplus ((\vec{x} \wedge (\underline{0}^b\underline{1}^b)^\infty) \downarrow k)$;

C.2.3. $\texttt{Collect}(\vec{x}, b)$

C.3. Repeat $\log q$ times, $\texttt{Spread}(\vec{x}, b)$; $b := 2b$.

C.4. Return $\vec{x}$.

Step C.2.2 has the effect that it reduces each key to its first half, leaving the second half free. The subsequent `Collect` pack the keys, so that all the free second halves get exploited. Consequently, after each iteration of the main loop, the number of words in consideration is halved, and hence so is the running time of each iteration. Thus, the cost of the first iteration dominates the cost of all other iterations, and the cost of the first iteration is $O(1 + |\vec{x}|/w)$. The point of the loop of step C.3 is just to restore the order of the keys, so that the signatures are output in the same order as the keys came in. In the following, we will often "forget" to discuss the details of restoring order. Since no key takes up more than one word, we conclude:

Lemma 7 *We can compute all signatures in linear time.* ■

Now, we sort the list $L =< (\texttt{Sign}(x_i), i) | x_i \in X >$. Each element $(\texttt{Sign}(x_i), i)$ is of length $O(w/(\log n \log\log n) + \log n) = O(w/(\log n \log\log n))$, so by Lemma 5, the sorting is done in linear time. Thereby we group a words in X with common signatures. Set $r = 3q(\log n) = O((\log n)^2(\log\log n))$. By Lemma 6 with probability at least $1 - 1/n$, there is no pair (x, y) in any group with $\|x \oplus y\|_1 > r$. For each such group Y, we pick an arbitrary element $y_0 \in Y$. Then no word in $Y \oplus y_0$ has more than $r = O((\log n)^2 \log\log n)$ $\underline{1}$s.

Cluster bustering

For cluster bustering, we are given a set Z $(= Y \oplus y_0$ from above) of keys with few $\underline{1}$s. Our general technique is to construct a fast random function f from keys with few $\underline{1}$s to short keys such that

- for any given key with few $\underline{1}$s, f errors with probability $< 1/2$.

- f is 1-1 for all keys for which it is successful.

Thus f will successfully reduce the length of at least half the elements in Z. For the remaining half, we recursively apply a new random function. Due to the halving, the overall complexity is asymptotically the same as that of applying one random function to all the elements.

More specifically, we consider each key divided into $\log n \log\log n$ equal sized segments, each of which is divided into r^2 equal sized pieces; making up for a total of $q = r^2(\log n)(\log\log n)$ pieces of length $k = w/q$.

Our compression will aim at "hashing" the non-empty pieces (i.e. pieces containing $\underline{1}$s) of a key into the beginning of the key. To make such a function injective, before the hashing, we give each non-empty piece a label identifying its position in the word. This is done as follows at constant cost.

$$x := \texttt{Spread}(x,k) \vee (\texttt{Mask}(\texttt{Spread}(x,k), 2k) \wedge \prod_{i=0}^{|x|/k-1} (\underline{0}^k \texttt{str}(i,k))$$

Above, $\prod_{i=0}^{|x|/k-1}(\texttt{0}^\texttt{k}\texttt{str(i,k)})$ is only computed once in time $O(|x|/k) = O(\log n)$, whish is less than a constant per key. Note that $\log i = \log\log n$ while $k = (\log n)^{\omega(1)}$ by (1), so we have no problems representing i by $k-1$ bits. Also note that $\texttt{Spread}(x,k)$ actually permutes the pieces, but since permutations are injective, this does not conflict with our purpose. The labeling doubles the length of the keys and of the pieces (set $b := 2b$ and $k := 2k$), but this does not affect our asymptotic running times. Now a key is uniquely identified by its set of non-empty pieces.

We will now be more specific about the hashing of the non-empty segments. For each segment independently, make a random cyclic shift with a multiple of k positions. We consider it an error if any two non-empty pieces end up at the same offset within their segment. Note that there are at most r non-empty pieces, and that there are r^2 different offsets in each segments. Thus, the probability that any two non-empty pieces end at the same offset is at most $1/r^2$, so the probability that any two non-empty pieces collide is at most $\binom{r}{2}/r^2 < 1/2$. If we do not have an error, we complete the compression by taking the bit-wise disjunction of all the $\log n \log\log n$ segments in time $O(\log\log n)$. The resulting key has the same length as that of a segment, which is $w/(\log n \log\log n)$. Lemma 5 now allows us to complete identifying duplicates by sorting in linear time.

We now need to specify how we will represent the random shifts, and how to compute them efficiently. For the representation, we just use one random string β of the same length as the keys. We will use the last $\log r^2$ positions in the ith segment of β to determine how much to shift the ith segment of x. This is done using the function `MultiShift` below.

Algorithm D: $\texttt{MultiShift}(\beta, x, s, k)$. Takes each segment $x\langle i\rangle_s$ and shift it cyclically to the right by k times the number of positions represented by the first $\log(s/k)$ bits of $\beta\langle i\rangle_s$.

D.1. For $j := \log k$ to $\log s - 1$ do to :

D.1.1. $\gamma := \texttt{Mask}(\beta \wedge (\underline{0}^{s-1}\,\underline{1})^\infty, s)$.

D.1.2. $x_1 := (x \wedge \gamma) \uparrow 2^j$.

D.1.3. $x_2 := (x_1 \wedge (\underline{0}^{2^j}\,\underline{1}^{s-2^j})^\infty) \vee ((x_1 \wedge (\underline{1}^{2^j}\,\underline{0}^{s-2^j})^\infty) \downarrow s)$.

D.1.4. $x := (x \wedge (\gamma \oplus \underline{1}^{\infty})) \vee x_2$.

D.1.5. $\beta := \beta \uparrow 1$.

Given a list $\vec{x} = x_1 \cdots x_n$ be a list of b-bit keys, $\vec{y} = \texttt{MultiShift}(\beta^n, \vec{x}, s, k)$ is computed in time $O(n(b/w)\log(s/k))$. Finally we need

Algorithm E: Combines the segments in each key in $\vec{y}$, thus reducing the length to s. If a key has two non-empty pieces at the same offset, error is notified by turning the key into all $\underline{1}$s.

E.1. While $b > s$ times:

E.1.1. $b := b/2$;

E.1.2. $\vec{y}_0 := (\vec{y} \wedge (\underline{1}^b \underline{0}^b)^{\infty})$; $\vec{y}_1 := (\vec{y} \wedge (\underline{0}^b \underline{1}^b)^{\infty}) \downarrow s$;

E.1.3. $\vec{y} := \vec{y}_1 \vee \vec{y}_2 \vee \texttt{Mask}(y_0 \wedge \texttt{Mask}(y_1, k), s)$

E.1.4. $\texttt{Collect}(\vec{y}, b)$

E.2. Repeat $\log q$ times, $\texttt{Spread}(\vec{y}, b)$; $b := 2b$.

As algorithm C, this algorithm runs in linear time due to the halving. Thus, we conclude:

Lemma 8 *Let $\vec{x} = x_1 \cdots x_n$ be a list of b-bit keys, each with at most $r = O((\log n)^{O(1)})$ $\underline{1}$s. In expected time $O(n + n(b/w)\log\log n)$, we can injectively reduce half the keys to length $w/(\log n \log\log n)$.* ∎

Together with Lemma 5, this completes the proof of Theorem 3 under our assumption (1).

5 Short words

In this section, we assume that (1) is false, i.e. that $w = (\log n)^{\Theta(1)}$. Note that if $b = O(\log n)$, constant time per key can be achieved using radix sort with character length $\log n$:

Lemma 9 *We can sort n $O(\log n)$ bit keys in linear time $O(n \log n)$.* ∎

Thus, assume $b = \omega(\log n)$. Our approach in this section goes as follows. First, we make some improved signatures, allowing us to collect keys in Hamming balls with radius $r = O(\log n/\log b) = O(\log n/\log\log n)$, as opposed to the radius of $O(\log n)$ in the previous section. We then show how to compress keys with r $\underline{1}$s to a length of $O(r \log r) = O(\log n)$ as opposed to the $O(r^2)$ in the previous section. Then radix sort completes the task. Our approach heavely uses that $\log b = O(\log\log n)$ because (1) is false.

Let k be an odd number deviating from $b \log b/\log n$ be at most 1. Here b is the current key length. Let $l = 3\log n$. For $i = 0, \ldots, l-1$, let $H(i)$ be a set of k independent samples from $\{0, \ldots, b-1\}$. Define the signature $\texttt{Sign}^H(x)$ of length l such that for $j = 0, \ldots, l-1$,

$$\texttt{Sign}^H(x)[i] = \bigoplus\{x[j] | j \in H(i)\}.$$

In order to prove that this is a good signature, we shall use

Lemma 10 ([AMRT96]) *Suppose that $x_1, x_2, \ldots, x_k$ are 0-1 variables which take value 1 with probability p. Then $\Pr(\bigoplus_{i=1}^{k} x_i = 0) = \frac{1}{2}(1 + (1-2p)^k)$, which is $\leq \max\{1 - pk/2, 1/2\}$ for k odd.*

Lemma 11 *For H randomly chosen, with probability $1 - 1/n$, for any two keys x and y in X, $|X| = n$, if $\|x \oplus y\|_1 \geq b/k = O(\log n/\log b)$, $\mathtt{Sign}^H(x) \neq \mathtt{Sign}^H(y)$.*

Proof: Let x and y be two keys with $\|x \oplus y\|_1 \geq b/k$. Set $z = x \oplus y$. Then $\mathtt{Sign}^H(z) = \mathtt{Sign}^H_\alpha(x) \oplus \mathtt{Sign}^H_\alpha(y)$. We are searching the probability that $\mathtt{Sign}_H(z) = \underline{0}^k$. For random j, the probability p that $z[j] = \underline{1}$ is $\|z\|_1/|z| = r/b$. Moreover, by Lemma 10, the probability that $\mathtt{Sign}^H(z)[i] = \underline{0}$ is $\leq \max\{1 - pk/2, 1/2\}$. But since $r \geq b/k$, $pk/2 = r/b \cdot k/2 \geq 1/2$. Hence $\max\{1 - pk/2, 1/2\} = 1/2$. Consequently, $\Pr(\mathtt{Sign}^H(z) = \underline{0}^k) \leq 2^{-l} = n^{-3}$. The lemma now follows from the fact that there are only n^2 pairs of keys. ∎

In order to implement $\mathtt{Sign}^H$ efficiently, we will use matrix transposition. A list of q words $x_1, \ldots, x_q$ are seen as representing p $(q \times q)$-matrices $A_1 \ldots, A_p$ as follows. $x_i\langle a\rangle_{/p}$ is the ith row of matrix A_a. Consequently, the entries of the matrices are of length $w/(pq)$, and entry (i, j) of A_a is found as $x_i\langle a\rangle_{/p}\langle j\rangle_{/q}$. Recall that transposing a matrix means swapping each entry (i, j) with entry (j, i). By transposing the above row of matrices, we mean constructing words $y_1, \ldots, y_q$ such that for any a, i, j, $y_i\langle a\rangle_{/p}\langle j\rangle_{/q} = x_j\langle a\rangle_{/p}\langle i\rangle_{/q}$.

Lemma 12 *We can transpose a row of $(q \times q)$-matrices in time $O(q \log q)$.*

Proof: Let $T(q)$ be time it takes to transpose a row of $(q \times q)$-matrices. Then $T(1) = 0$. For $q > 1$, we simply use the recursive definition of transposition. Divide each matrix into quadrants, giving us two rows of $(q/2 \times q/2)$-matrices. These two rows are then transposed in time $2T(q/2)$. Having transposed all quadrants, to finalize the transposition, for each matrix, we just need to swap quadrant $(1, 1)$ with quadrant $(2, 2)$. This is implemented as follows. For $i = 1, \ldots, q/2$, $y_i = (x_i \wedge (\underline{0}^{q/2}\,\underline{1}^{q/2})^\infty) \wedge ((x_{i+q/2} \wedge (\underline{0}^{q/2}\,\underline{1}^{q/2})^\infty) \downarrow q/2)$, and for $i = q/2 + 1, \ldots, q$, $y_i = (x_i \wedge (\underline{1}^{q/2}\,\underline{0}^{q/2})^\infty) \wedge ((x_{i-q/2} \wedge (\underline{1}^{q/2}\,\underline{0}^{q/2})^\infty) \uparrow q/2)$. Thus, swapping the quadrants is done in time $O(q)$. Thus $T(q) \leq 2T(q/2) + q$, so $T(q) = O(q \log q)$. ∎

A similar matrix transposition is used in [PS76] for unbounded word-lengths.

Lemma 13 *We can calculate $\mathtt{Sign}^H(x)$ in $O(1 + b/w \cdot \log b)$ amortized cost per key.*

Proof: Consider words $\vec{x}_1, \ldots, \vec{x}_b$, each consisting of w/b keys. The words are viewed as a row of $b \times b$ matrices. Each row in a matrix is a key. Transpose the matrices in time $O(b \log b)$. Let $\vec{y}_1, \ldots, \vec{y}_b$ be the resulting words. Now $\vec{y}_j$ contains bit j of every single key.

For $i := 1, \ldots, l$, set $\vec{z}_i = \bigoplus_{j \in H(i)} \vec{y}_j$. This takes time $O(\sum_i |H(i|) = O(b \log b)$. Transpose $\vec{x}_1, \ldots, \vec{x}_b$ view as a row of $l \times l$ matrices, in time $O(l \log l)$. The signatures of our original keys are now found as the rows of the created vertices. It is straightforward to locate the signature of any specific key in constant time. The number of keys involved is $bw/b = w$, so the amortized cost per key is $O(1 + b/w \cdot \log b)$, as desired. ∎

The keys are now clustered. As in Section 4, for each group Y, we pick an arbitrary element $y_0 \in Y$. Then the maximal number of $\underline{1}$s in any key in $Y \oplus y_0$ is $r = O(\log n/\log b)$. Note that this is optimal, in the sense that there are less than n b-bit keys with $\log n/\log b$ $\underline{1}$s.

Cluster bustering for short words

We are considering n b-bit keys, each containing at most $r = \log n/\log b$ $\underline{1}$s. Consider each key as divided into pieces of lenght $k = \log b$, where b is the key lenght. As in Section 4, we give each non-empty piece a label identifying its position in the key. Each label takes $\log b$ bits, so this doubles the piece and key length. Now each key is uniquely identified by its set of $\leq r$ non-empty segments. As in Section 4, we wish to compress the non-empty pieces into the beginning of the keys, but this time we want a much tighter compression.

First choose a random permutation π of the pieces in a key. Divide $\pi(x)$ into $q = r/(3\log r)$ segments. In each segment, we expect $\leq 3\log r$ non-empty pieces. Using Chernoff bounds (see e.g. [Pap94]), it follows that the probability that a segment has more than $6\log r$ non-empty pieces is $< 1/r$. Actually our distribution is not quite Binomial distribution. It would be if each non-empty piece was independently assigned a random segment. However, the dependency introduced by all the segments containing the same number of pieces means that the probability of deviation goes down in the tail, so the use of Chernoff bounds is still valid. We consider it an error if any segment contains more than $6\log r$ non-empty pieces, so it follows, that the probability of error for a given key $\pi(x)$ is $< q/r = O(1/\log r) < 1/2$ for r large.

In the following we assume that there is no error. We say a segment is *packed* if all the non-empty pieces lie in the beginning, i.e. if the ith non-empty piece is the ith piece. We now pack segments of exponentially increasing sizes as follows.

- First, consider each piece a segment. Then trivially, each segment is packed.
- Repeat the following for $\log(6\log r)$ rounds: take each even numbered segment and unite with it neighboring segment to the right. To pack the united segment, take the packed non-empty pieces of the right half and shift them to the left so that they become juxtaposed with the packed non-empty pieces of the left part. At the end, each segment is packed and of size $6\log r \cdot k$.
- Repeat the following for $\log(b/q) - \log(6\log r)$ times: as above, take each odd numbered segment and pack its non-empty pieces into its neighboring segment to the left. Since the segments have room for $6\log r$ non-empty pieces, there will be no non-empty pieces remaining in any of the odd numbered segment. Hence all the odd numbered segments may be removed. As a result, the length of the key is halved, and this will be crucial for the overall running time.
- At the end each segment contains $6\log r$ pieces, and we have $q = r/(3\log r)$ segments. Thus, we have compressed all the non-empty pieces in a total of $2r$ pieces. The length of a piece is $O(\log b)$, so the length of the compressed key is $O(r\log b) = O(\log n)$, as desired.

It follows as a special case of Lemma 13, that we can compute $\pi(x)$ for all keys x in time $O(n(1 + b/w \cdot \log b))$. In order to find out how much to shift in connection with the packing, we need a function `MultiCount` that is a kind of reverse to `Multishift`.

Algorithm F: $\texttt{MultiCount}(x, s, k)$. Outputs β such that for each segment $x\langle i\rangle_s$, $\texttt{int}(\beta\langle i\rangle_s)$ is the largest number d such that $x\langle i\rangle_s\langle d\rangle_k$ is non-empty.

F.1. $\beta := \underline{0}^\infty$.

F.2. For $j := \log s$ down to $\log k$:

F.2.1. $x_1 := x \wedge (\underline{0}^{2^j-k}\,\underline{1}^{s-2^j+k})^\infty$.

F.2.2. $\gamma := \texttt{Mask}(x_1, s)$.

F.2.3. $\beta := (\beta \vee (\gamma \wedge (\underline{0}^{s-1}\,\underline{1})^\infty) \downarrow 1$.

F.2.4. $x := (x \wedge (\gamma \oplus \underline{1}^\infty)) \vee (x_1 \downarrow 2^j)$.

F.3. Return β.

We are now ready to describe the compression.

Algorithm G: $\texttt{Compress}(\vec{x})$ where $\vec{x}$ is a packed list of keys $\pi(x)$, each of length b. If there is an overflow ($\geq 6\log r$ non-empty pieces) in a segment, it is turned into all $\underline{1}$s.

G.1. $s := k$;

G.2. While $s < b$:

G.2.1. $\vec{x}_0 := \vec{x} \wedge (\underline{1}^s\,\underline{0}^s)^\infty$;

G.2.2. $\vec{x}_1 := \vec{x} \wedge (\underline{0}^s\,\underline{1}^s)^\infty$;

G.2.3. $\beta := \texttt{MultiCount}(\vec{x}_0, 2s, k)$;

G.2.4. $\vec{x} := x_0 \vee \texttt{MultiShift}(\beta, \vec{x}_1 \downarrow (s-1), 2s, k)$;

G.2.5. If $s < 6k\log r$, $s := 2s$;

G.2.6. Otherwise,

G.2.6.1. $\vec{x} := \vec{x} \vee (\texttt{Mask}(\vec{x} \wedge (\underline{0}^s\,\underline{1}^s)^\infty, 2s)$ — marking overflow.

G.2.6.2. $b := b/2$;

G.2.6.3. $\vec{x} := (\vec{x} \wedge (\underline{1}^b\underline{0}^b)) \vee ((\vec{x} \wedge (\underline{0}^b\underline{1}^b)) \uparrow s)$;

G.2.6.4. $\texttt{Collect}(\vec{x}, b)$;

G.3. Return $\vec{x}$.

Step G.2.6.3 permutes the non-empty segments within the keys, but clearly this does not affect the injectivity of the compression. For the first rounds with $s < 6k\log r$, the time bound is $O(n(b/w)$ $(\log(6\log r))^2) = O(n(b/w)(\log\log\log n)^2)$. Due the halving, the complexity of the remaining rounds is $O(\log\log\log n)$. That is, the compression is done in $O(n(b/w)(\log\log\log n)^2) = O(n(b/w)\log\log n)$ time. This completes the proof of Theorem 1.

6 Batched universal hashing

In this section, we briefly describe some other results that can be achieved with similar techniques. In [BMM96] it is argued that Shönhage-Strassen multiplication can be simulated on the Practical RAM to run in time $O((\log w)(\log\log w)^2)$. In fact, they get that if a word consists of consecutive pairs of k-bit integers, we can multiply all pairs in time $O(\log k)(\log\log k)^2)$. Combining this with the techniques from above and the multiplicative strongly universal hashing from [Die96], one can implement batched strongly universal hashing of $n \geq w$, w-bit integers into k-bit integers in time $O(n \log k (\log\log k)^2)$, that is, at cost $O(\log k(\log\log k)^2)$ per hashed integer. In contrast it follows from [AMRT96], that if we want to hash integers one at the time, the cost is $\Omega(\sqrt{k/\log k})$ on the more general AC^0 RAM, even if we allow pre-processed tables of size $2^{k^{O(1)}}$.

This approach leads to an $O(n \log\log n (\log\log\log n)^2)$ sorting algorithm, which is worse than the bound in Theorem 1. Also, it is more impractical since implementing Shönhage-Strassen on the Practical RAM would be very complicated. However, the idea can be taken further to derive a Practical RAM priority queue supporting `insert` and `extract-min` in $O(\log\log n(\log\log\log n)^)$ expected time per operation, that is, at a slight cost in the complexity, we avoid the amortization and monotonicity of Corollary 4.

What could make the hashing ideas practically relevant was if our processor was like a vector processor with hardware support for locally multiplying pairs of k-bit integers, $k \ll w$, that is, if hardware replaced Shönhage-Strassen above, then this could potentially give us very fast sorting and priority queues for up to 2^k keys.

References

[AH92] S. Albers and T. Hagerup, Improved parallel integer sorting without concurrent writing, *in Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 463–472, 1992.

[AHNR95] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proc. 27th ACM Symposium on Theory of Computing (STOC)*, pages 427–436, 1995.

[AMRT96] A. Andersson, P.B. Miltersen, S. Riis, and M. Thorup. Static Dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient (see `http://www.diku.dk/~mthorup`). *FOCS'96*, to appear.

[BH89] P. Beame and J. Håstad. Optimal bounds on the decision problems on the CRCW PRAM, *J. ACM* **36** (1989), 643–670.

[BMM96] A. Brodnik, P.B. Miltersen, and I. Munro. Word level parallelism - some upper and lower bounds, Manuscript, 1996.

[CW79] J.L. Carter and M.N. Wegman. Universal classes of hash functions, *J. Comp. Syst. Sci.* **18** (1979), 143–154.

[Die96] M. Dietzfelbinger. Universal classes of hash functions, In *Proc. STACS'96, LNCS 1046,* pages 569–580, 1996.

[FW93] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.

[Mil96] P.B. Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. *ICALP'96*, to appear.

[Pap94] C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[PS76] Pratt and Stockmeyer. A Characterization of the Power of Vector Machines. *JCSS*, 12:198–221, 1976.

[Ram96] R. Raman. Fast Algorithms for Shortest Paths and Sorting. *ESA'96*, to appear.

[Tho96] M. Thorup. On RAM priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.