UPPER BOUNDS FOR SORTING INTEGERS ON RANDOM ACCESS MACHINES

by

David Kirkpatrick*
Stefan Reisch**

Technical Report 81-12***

September, 1981

---

* Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada

** Fakultät für Mathematik, Universität Bielefeld, Federal Republic of Germany

*** This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, grant A3583.

Abstract

Two models of Random Access Machines suitable for sorting integers are presented. Our main results show that i) a RAM with addition, subtraction, multiplication, and integer division can sort n integers in the range $[0,2^{cn}]$ in $O(n \log c + n)$ steps; ii) a RAM with addition, subtraction, and left and right shifts can sort any n integers in linear time; iii) a RAM with addition, subtraction, and left and right shifts can sort n integers in the range $[0,n^c]$ in $O(n \log c + n)$ steps, where all intermediate results are bounded in value by the largest input.

1. Introduction

The general 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 restrict one's attention to comparison-based algorithms. In this setting, it is well-known that O(n log n) operations (comparisons) are necessary and sufficient to sort an arbitrary n-tuple. It is also well-known that if an n-tuple consists entirely of integers in the range [0,n-1] (or, in fact, any range of size O(n)), then it can be sorted in O(n) operations by a RAM. This can be done by exploiting the capability of indirect addressing. In this context the question arises whether the capability of arithmetic operations will help to speed up sorting processes even if the integers to be sorted are very large compared to n. A first answer to this question can be given by generalizing the above cited linear time algorithm. Applying a multi-pass bucket-sort one gets an O(n m) upper bound for sorting an n-tuple of integers in the range $[0,n^m-1]$ [1]. It is natural to ask what is the largest range of integers for which a sorting algorithm can be constructed that runs asymptotically faster than the conventional O(n log n) comparison-based algorithms. We will describe some sorting algorithms for two models of RAM's, which extend the range of integers over which comparison based sorting can be improved.

2. Models of computation

We consider two variants of the RAM-model which is described in [1] 5 pp. Inputs to our RAMs are n-tuples $(x_1,\ldots,x_n) \in N^n$ which are written on an input tape. The output, which is some permutation of the input n-tuple, is written onto an output tape. Both of our models include conventional branching instructions and the ability to indirectly address arbitrarily many registers. Our two RAMs are distinguished according to their ability to execute specific integer operations. A machine of type RAM_i $(i = 1,2)$ is able to carry out integer operations from the set O_i only:

$$O_1 = \{+,\dot{-},\times,\lfloor/\rfloor\}$$

$$O_2 = \{+,\dot{-},\text{Shift L},\text{Shift R}\}$$

Let c(a) denote the content of register a. The shift instruction Shift R (a) (respectively, Shift L(a)) assigns to the accumulator (i.e. register 0) the value $\lfloor c(0)/2^{c(a)} \rfloor$ (respectively, $c(0)2^{c(a)}$), that is, its previous content is shifted right (respectively left) by c(a) bits. "$\dot{-}$" denotes the monus operation, i.e. $x\dot{-}y = \max\{0,x-y\}$.

At a first glance the operation set O_2 appears to be weaker than the set O_1, since shifts involve very restricted types of integer multiplication and division. In fact, this is the case provided the size of the operands and results does not become too large. However, if shifts are used in a completely unrestricted fashion, then they introduce the power of exponentiation and hence operation set O_2 becomes more directly comparable with the set $\{+,\dot{-},\times,\lfloor/\rfloor,2^x\}$.

We can make the above comparisons more precise by introducing two schemes for charging for operations on our different models. First, it is worth pointing out that traditionally sorting has been studied on unit charged (i.e. unit cost criterion) models of computation. Certainly, this assumption underlies the basic results cited in the preceding section. To be consistent, then, it makes sense to assign a unit cost to the reading, printing, or other manipulation of integers comparable in size to the largest input I (a not unreasonable choice when the size of I is comparable to the word size of real machines). If this unit charging policy is extended to the manipulation of integers of all sizes we have what we call the uniform charging scheme. On the other hand, we can account, at least in part, for the higher cost of multiprecision operations by charging in proportion to $\log_I(x)$ for the manipulation of an integer x. We call this a logarithmic charging scheme. While logarithmic charging is clearly the more realistic policy, it is one of the objectives of this paper to shed light on the inherent uniform complexity of sorting (and, in particular, to complement established lower bounds using this measure [8]).

Let $S_i(n,R)$ (respectively, $S_i^*(n,R)$), i = 1,2, denote the time required, assuming uniform (respectively, logarithmic) charging, by a RAM_i to sort an arbitrary set of n integers drawn from the range [0,R-1]. (Since $S_i(n,R)$ and $S_i^*(n,R)$ are both $\Omega(n)$, it should be clear that the specific range [0,R-1] is irrelevant and that $S_i(n,R)$ and $S_i^*(n,R)$ denote equally well the complexity of sorting any n integers $x_1,\ldots,x_n$ where $\max\{x_1,\ldots,x_n\}-\min\{x_1,\ldots,x_n\} < R$.)

In the next section we develop a RAM_1-algorithm that, assuming uniform charging, significantly extends the range of integers over which conventional

sorting algorithms can be asymptotically improved. While the algorithm fully exploits uniform charging, using integers significantly larger than the inputs, the result is interesting from a theoretical point of view, since it suggests that the lower bound arguments of Paul and Simon [8] for sorting on RAM's without integer-division cannot be generalized in a straightforward way. In section 4, we describe an improved radix sorting algorithm that can be performed on both RAM models. Since this algorithm uses integers of the size of the largest input only, its uniform and logarithmic complexities are equivalent. The algorithm employs a technique of systematic range reduction which also can be used to improve the result of section 3. In section 5 we present an integer sorting algorithm for RAM_2's that runs in linear time under the uniform charging scheme. Once again this result has more theoretical than practical importance since the algorithm uses extremely large integers. The concluding section 6 comprises a discussion of related work. In particular, we give some comments on the $\Omega(n \log n)$ lower bound due to Paul and Simon [8] for RAM's with addition, subtraction, and multiplication, and we discuss the question of whether integer-division can help to speed up sorting. For understanding this section it is useful to be familiar with the proof of Paul and Simon.

3. Fast sorting using multiplication and integer-division

As we observed in section 1, bucket-sorting techniques yield an $O(nm)$ upper bound for sorting an n-tuple of integers in the range $[0,n^m-1]$. Since this algorithm can be implemented on both of our models without introducing integers larger than the maximum input, we have

$$S_i^*(n,n^m) \leq O(n\,m) \text{ for } i = 1,2.$$

If we are interested in sorting with uniform charging this bound can be strengthened significantly.

In order to get a faster sorting algorithm we will consider first the following algorithm which has the disadvantage of working correctly only for "spread out" inputs. This will be made precise in the succeeding Lemma.

Algorithm 1.

Step 1: Read input $x_1,\ldots,x_n$

Step 2: Compute $p_n = \sum_{i=1}^{n} i\, x_i$

Step 3: Compute $q_n = \sum_{i=1}^{n} x_i$

Step 4: For $i = n,\ldots,1$ do

4a: $j_i \leftarrow \left\lfloor \frac{p_i}{q_i} \right\rfloor$

4b: $x_{\pi(i)} \leftarrow x_{j_i} + (x_{j_i+1} \dot{-} x_{j_i})$ $(=\max(x_{j_i},x_{j_i+1}))$

4c: Output $x_{\pi(i)}$

4d: $$\pi(i) \leftarrow \begin{cases} j_i & \text{if } x_{\pi(i)} = x_{j_i} \\ j_{i+1} & \text{if } x_{\pi(i)} = x_{j_i+1} \end{cases}$$

4e: $$p_{i-1} \leftarrow p_i \div \pi(i)x_{\pi(i)}$$
$$q_{i-1} \leftarrow q_i \div x_{\pi(i)}.$$

Lemma 1

(i) Algorithm 1 has uniform time complexity O(n).

(ii) Algorithm 1 sorts an input $(x_1,\ldots,x_n)$ correctly if:

$$\forall_{\substack{i,j(1\le i,j\le n)\\ i\neq j}} \frac{x_i}{x_j} \notin (\tfrac{1}{n-1}, n-1).$$

Proof: The validity of the first statement is obvious. To prove part (ii) first note that

$$\left\lfloor \frac{\sum_{j=1}^{i} \pi(j)x_{\pi(j)}}{\sum_{j=1}^{i} x_{\pi(j)}} \right\rfloor = \left\lfloor \pi(i) + \frac{\sum_{j=1}^{i-1} (\pi(j)-\pi(i)x_{\pi(j)}}{\sum_{j=1}^{i} x_{\pi(j)}} \right\rfloor .$$

Since $x_{\pi(i-1)} \le \frac{1}{n-1} x_{\pi(i)}$ we have

$$x_{\pi(j)} \le (\tfrac{1}{n-1})^{i-j} x_{\pi(i)} \quad \text{for} \quad i > j$$

and

$$\sum_{j=1}^{i-1} (|\pi(j)-\pi(i)|)-1)x_{\pi(j)} \le (n-2) \sum_{j=1}^{i-1} x_{\pi(j)} \le (n-2)x_{\pi(i-1)} \sum_{j=0}^{i-2} \frac{1}{(n-1)^j}$$

$$< \frac{n-2}{1-\frac{1}{n-1}} x_{\pi(i-1)} = (n-1)x_{\pi(i-1)} \le x_{\pi(i)}.$$

Thus $$\left| \sum_{j=1}^{i-1} (\pi(j)-\pi(i))x_{\pi(j)} \right| < \sum_{j=1}^{i} x_{\pi(j)}$$

which implies

$$\left\lfloor \frac{\sum_{j=1}^{i} \pi(j)x_{\pi(i)}}{\sum_{j=1}^{i} x_{\pi(j)}} \right\rfloor \in \{\pi(i),\pi(i)-1\}. \qquad \square$$

The requirement (ii) of Lemma 1 means that each input-integer is (n-1)-times larger than any smaller input-integer. This very restrictive requirement trivially can be weakened if integers $x_1^d,\ldots,x_n^d$ are sorted instead of input $x_1,\ldots,x_n$.

Lemma 2: Let $x_1,\ldots,x_n \in N$ satisfy $\frac{x_i}{x_j} \notin (\frac{d}{d+1},\frac{d+1}{d})$ for $i \neq j$. Then for

$$y_1 = x_1^{d\lceil \log n \rceil},\ldots,y_n = x_n^{d\lceil \log n \rceil}$$

$$\frac{y_i}{y_j} \notin (\frac{1}{n-1},n-1) \quad \text{for} \quad i \neq j.$$

Proof: Suppose $y_i > y_j$. Then,

$$\frac{y_i}{y_j} = (\frac{x_i}{x_j})^{d\lceil \log n \rceil} > (\frac{d+1}{d})^{d\lceil \log n \rceil} > 2^{\lceil \log n \rceil} > n-1. \qquad \square$$

Hence it will be useful to have a fast algorithm computing the function $h: (N^n \times N) \to N^n$ with $h((x_1,\ldots,x_n),c) = (x_1^{2^c},\ldots,x_n^{2^c})$. We now proceed to the description of such an algorithm whose existence is interesting in its own right.

An essential tool for the principal construction of our algorithm is Chinese Remaindering:

Chinese Remainder Theorem

Let $\{p_1,\ldots,p_n\}$ be a set of positive integers which are pairwise relatively prime. Let

$$p = \prod_{i=1}^{n} p_i$$

and

$$(z_1,\ldots,z_n) \in N^n \quad \text{with} \quad 0 \leq z_i < p_i \quad (\text{for } 1 \leq i \leq n).$$

Then there exists exactly one $z \in N$ satisfying:

(i) $0 \leq z < p$; and

(*)

(ii) $\forall_{i(1\leq i\leq n)} z_i \equiv z \bmod p_i$.

An integer z with property (*) for two given n-tuples $(z_1,\ldots,z_n)$ and $(p_1,\ldots,p_n)$ can be computed in the following way: For all $i(1 \leq i \leq n)$ let $q_i = \frac{p}{p_i}$. Because of the relative primality of $p_1,\ldots,p_n$ there exists an $r_i \in N$ with $1 \leq r_i < p_i$ and $q_i r_i \equiv 1 \bmod p_i$.

Then it is easy to see that

$$z = \left(\sum_{j=1}^{n} q_j r_j z_j\right) \bmod p$$

has required property (*) since

$$\sum_{j=1}^{n} q_j r_j z_j \equiv q_i r_i z_i \equiv z_i \bmod p_i$$

(for this method to obtain z see [1] p. 294 lemma 8.2).

In order to apply Chinese Remaindering we have to find suitable integers $p_1,\ldots,p_n$, which are mutually prime. The next Lemma shows that this can be done in a straightforward manner:

Lemma 3: Let $p_1 \in N$ be arbitrary and let

$$p_i = \left(\prod_{j=1}^{i=1} p_j\right) + 1 \quad \text{for} \quad i = 2,\ldots,n, \text{ and}$$

$$p = \prod_{i=1}^{n} p_i; \quad q_i = \frac{p}{p_i}, \quad r_i = p_i - 1 = \prod_{j=1}^{i-1} p_j. \quad \text{Then}$$

(i) $p_1,\ldots,p_n$ are pairwise relatively prime

(ii) $q_i r_i \equiv 1 \bmod p_i$.

Proof: The validity of proposition (i) is obvious. Furthermore, since

$$q_i = \prod_{\substack{j=1 \\ j\neq i}}^{n} p_j = \left(\prod_{j=1}^{i-1} p_j\right)\left(\prod_{j=i+1}^{n} p_j\right)$$

and $p_j \equiv 1 \bmod p_i$ for $j > i$, it follows that

$$q_i \equiv \left(\prod_{j=1}^{i-1} p_j\right) \bmod p_i$$

$$\equiv -1 \bmod p_i$$

Proposition (ii) now follows immediately. □

The following algorithm computes the function

$h: (N^n \times N) \to N^n$ with $h((x_1,\ldots,x_n),c) = (x_1^{2^c},\ldots,x_n^{2^c})$.

Algorithm 2

Step 1: Read input $x_1,\ldots,x_n$

Step 2: Compute $\bar{x} = \max_{1\le i<n} (x_1,\ldots,x_n)$

Step 3: 3a: $p_1 \leftarrow \bar{x}^{2^c}$

3b: $r_1 \leftarrow 1$

Step 4: For $i = 1,\ldots,n$ do

4a: $r_1 \leftarrow r_{i-1} \times p_{i-1} (= \prod_{j=1}^{i-1} p_j)$

4b: $p_i \leftarrow r_i + 1$

Step 5: Compute $p = \prod_{i=1}^{n} p_i$

Step 6: For $i = 1,\ldots,n$ do

$q_i \leftarrow \frac{p}{p_i}$

Step 7: Do 7a: $u' \leftarrow \sum_{i=1}^{n} q_i r_i x_i$

7b: $u \leftarrow u' \bmod p \; (= u' - \left\lfloor \frac{u'}{p} \right\rfloor p)$

Step 8: Compute $\omega: = u^{2^c}$

Step 9: For $i = 1,\ldots,n$ do

9a: $y_i \leftarrow \omega \bmod p_i \left(= \omega - \left\lfloor \frac{\omega}{p_i} \right\rfloor p_i \right)$

9b: Write output y_i.

In a first stage, including steps 1-6 the algorithm constructs integers $p_1,\ldots,p_n$ which are pairwise relatively prime and sufficiently large. In step 7 these integers are used to encode input integers $x_1,\ldots,x_n$ into an integer u. Step 8 computes the 2^c-th power of u and by step 9 we get numbers $y_i = x_i^{2^c}$. The preceeding considerations and the fact that for $x,y,p \in N$, $xy \equiv (x \bmod p)(y \bmod p) \bmod p$ show the correctness of algorithm 2.

It is easy to see that steps 1,2,3,4,5,6,7 and 9 require $O(n)$ steps. Performing steps 3 and 8 by continued squaring requires time $O(c)$. Hence, we have established the following:

Theorem 1. There is a RAM_1-algorithm computing the function

$$h: ((x_1,\ldots,x_n),c) \to (x_1^{2^c},\ldots,x_n^{2^c})$$

with uniform time complexity $O(n + c)$.

From theorem 1 and Lemma 1 we can deduce our first main result.

Theorem 2. There is an algorithm with uniform time-complexity $O(n + c)$ sorting n natural numbers $x_1,\ldots,x_n$ correctly, if the following condition is satisfied:

$$\forall_{i,j}(1 \leq i, j \leq n)\ (x_i > x_j) \Rightarrow \frac{x_i}{x_j} \geq 1 + \frac{1}{2^c}\ . \qquad (**)$$

Proof: Concatenate a modified version of algorithm 2 with algorithm 1. The algorithm runs in three stages:

1. First compute integers, $y_1,\ldots,y_n$ employing algorithm 2:

$$y_i = x_i^{(2^{c+\lceil \log_2 n \rceil})} \geq (x_i^{2^c})^{n\lceil \log n \rceil}.$$

Because of (**) then we have

$$\forall_{i,j(1\leq i,j\leq n)}\ (y_i > y_j) \Rightarrow \frac{y_i}{y_j} \geq \left[(1 + \frac{1}{2^c})^{2^c} \right]^n \geq 2^n. \qquad (***)$$

2. Compute $z_i = 2^{i-1} y_j$.
   Obviously the $z_i (1 \leq i \leq n)$ have the same order as the $x_i (1 \leq i \leq n)$ and $i \neq j \Rightarrow \frac{z_i}{z_j} \notin (\frac{1}{2},2)$.

3. Compute $z_1^{\lceil \log n \rceil},\ldots,z_n^{\lceil \log n \rceil}$ and sort these integers $z_1,\ldots,z_n$ by algorithm 1.

Stage 1 requires $O(c + \log n)$ steps, and stage 2 and stage 3 can be carried out in $O(n)$ steps. Hence the whole sorting algorithm has a running time of $O(n + c)$ steps.

Theorem 2 has one important corollary:

Corollary 1: Natural numbers $x_1,\ldots,x_n$ satisfying $x_i < 2^{cn}$ (for $i = 1,\ldots,n$) can be sorted within $O(cn)$ computation steps by a RAM_1, i.e. $S_1(n,2^{cn}) \leq O(cn)$.

Proof: If $x_1,\ldots,x_n \in [0,2^{cn} - 1]$ are input integers we have

$$x_i > x_j = \frac{x_i}{x_j} = 1 + \frac{x_i - x_j}{x_j} \geq 1 + \frac{1}{x_j} \geq 1 + \frac{1}{2^{cn}} .$$

Therefore $x_1,\ldots,x_n$ can be sorted in time $O(n + cn) = O(cn)$.

4. An improvement of radix-sort

In this section we will present a sorting algorithm which is applicable to both models of RAM's. The basic result of this section is a technique of systematic range reduction that is of interest in its own right.

Lemma 4: $S_i^*(n,2^k) \leq S_i^*(n,2^{\lceil k/2 \rceil}) + O(n + \log k), \quad i = 1,2.$

Before proving this Lemma we will show two important consequences. The first one is an improvement of corollary 1.

Corollary 2: $S_1(n,2^{cn}) \leq O(n \log c + n).$

Proof: For simplicity assume that $c = 2^t$ for some $t \in N$. By induction on t, Lemma 4 can be generalized to

$$S_i(n,2^{2^t n}) \leq S_i(n,2^n) + O(t(n + t))$$

Combining this with corollary 1, we have

$$S_1(n,2^{2^t n}) \leq O(t(n+t) + n).$$

Corollary 2 now follows immediately. □

The second corollary of lemma 4 describes a new radix-sorting method that increases the range over which conventional sorting algorithms can be improved, for both of our RAM models, even assuming logarithmic charging.

Corollary 3: $S_i^*(n,n^{m-1}) \leq O(n \log m + n)$

Proof: The result follows from lemma 4 by induction on m, exploiting the trivial bound $S_i^*(n,n) = O(n)$. □

Proof of Lemma 4: Suppose we are given n integers in the range $[0,2^k-1]$, for some $k \in N$. We will prove the result for RAM_2's only; the proof for RAM_1's involves a straightforward simulation of SHIFTing using ordinary multiplication and integer division (which entails an additive overhead of at most O(log k)). Furthermore, we assume that k is known in advance; its computation would require at most O(log k) steps on either model.

Imagine that we have a RAM_2 with $2^{\lceil k/2 \rceil} + O(n)$ registers each with a capacity of k bits (necessary and sufficient to represent numbers in the range $[0,2^{k-1}]$. Thus we can refer, without confusion to the "first" i bits of a specific register. (Of course these bits can be accessed by appropriate SHIFTing).

The algorithm starts with $2^{\lfloor k/2 \rfloor}$ empty buckets. Input integers are assigned, in succession, to buckets; the bucket number is given by the integer's first $\lfloor k/2 \rfloor$ bits (that is, its first half). An integer is represented within its bucket by its second $\lceil k/2 \rceil$ bits (which together with the bucket number uniquely identify the original integer). Obviously, when k is large most buckets will remain empty, however, each bucket must have the capacity for all n integers.

Suppose that some b buckets, we will refer to them as $B_1, B_2, \ldots, B_b$, are assigned one or more integers. (We call these "active" buckets). Let i_j be the index of B_j and let n_j be the number of elements assigned to B_j. The

bucketing procedure builds a list structure of size $2^{\lfloor k/2 \rfloor} + O(n)$ (see Figure 1) containing:

i) a list Bucket [i], for $0 \leq i \leq 2^{\lfloor k/2 \rfloor}$ of the elements assigned to bucket number i; and

ii) a list Active-buckets of b active bucket records, each of which gives the bucket number and size (number of elements) of one active bucket.

Since the buckets are assumed to be initialized, it should be clear that the bucketing stage can be completed in constant time per element, that is O(n) steps in total.

The algorithm next sorts the list Active-buckets <u>by size</u>, that is, we can assume that $n_1 \leq n_2 \leq \ldots \leq n_b$. This can be done using a conventional bucket sort since $b \leq n$ and $n_i \leq n$, for all i. Thus the original problem has been reduced to the following:

i) sort the list Active-buckets on the bucket numbers $(i_1, i_2, \ldots, i_b)$;

ii) sort the list of elements associated with each active bucket; and

iii) combine the sorted bucket lists in the order specified by the sorted Active-bucket list to give the final sorted list.

Clearly, step iii) can be accomplished in O(n) steps. Note that all of the numbers to be sorted in steps i) and ii) lie in the range $[0, 2^{\lceil k/2 \rceil} - 1]$.

Let r denote the unique integer satisfying

$$\sum_{j=1}^{r} n_j < b \quad \text{and} \quad \sum_{j=1}^{r+1} n_j \geq b.$$

Let $$n_0 = b - \sum_{j=1}^{r} n_j$$

and $n_* = \max\{n_0, n_r\}$. A new bucket, call it B_0, is formed by removing n_0 elements from bucket B_{r+1}. The resulting sequence of buckets $B_0,\ldots,B_b$ has the properties:

1. $1 \leq n_i \leq n_*$ for $0 \leq i \leq r$; and
2. exactly n-b elements lie in the union of buckets $B_{r+1},\ldots,B_b$.

The algorithm now,

(a) sorts bucket lists $B_0,B_1,\ldots,B_r$ individually, using any conventional ($O(n \log n)$) sorting technique;

(b) coalesces bucket lists $B_{r+1},\ldots,B_b$ along with the list of active bucket numbers into one large list with exactly n elements. (Along with each element is recorded the bucket number from which it came);

(c) sorts the coalesced list; and

(d) recovers the sorted individual lists from the result of step (c) (using the information recorded in step (b)).

Steps (b) and (d) are easily implemented in $O(n)$ steps. Step (c) is done recursively in $S(n,2^{\lceil k/2 \rceil})$ steps. Hence, to complete the proof, it suffices to show that step (a) can be done in $O(n)$ steps. As described, step (a) takes

$$O\left(\sum_{i=0}^{r} n_i \log n_i\right)$$

steps. We can assume that $n_* > 1$ (otherwise step (a) takes no time at all). By property (1) and straightforward maximization we have

$$\sum_{i=0}^{r} n_i \log n_i < (b-r)n_*$$

But, since $n_i \geq n_*$ for $i > r + 1$,

$$n \geq \sum_{i=0}^{r+1} n_i + (b-r-1)n_*$$

$$> (b-r)n_*,$$

and hence step (a) requires at most $O(n)$ steps. □

It should be noted that the analysis of the above algorithm cannot be tightened. If k is even then in either of the extreme cases, that is when all of the integers are placed in the same bucket, or each is placed in its own bucket, the recurrence

$$S(n,2^k) = S(n,2^{\lceil k/2 \rceil}) + O(n)$$

follows.

Our algorithm found it beneficial to "batch" together some of its sorting subproblems (see step (b)). It is interesting to note that while such batching is not advantageous for the general sorting problem, lower bounds like those of [8] suggest that, for integer sorting, this may be the most efficient approach to the solution of several (apparently independent) subproblems.

Careful inspection of the proof of lemma 4 reveals that it can (and perhaps should) be stated more precisely as follows:

Lemma 4': Assuming our RAM's have at least $2^{\lceil k/2 \rceil} + O(n)$ registers, each with a capacity of k bits,

$$S_i^*(n,2^k) \leq S_i^*(n,2^{\lceil k/2 \rceil}) + O(n + \log k), \qquad i = 1,2.$$

Obviously, the space requirements of the above algorithm will be prohibitive in many situations. However, our scheme for range reduction can be generalized in a straightforward way to apply to less well endowed RAM's as well. Specifically,

Lemma 5: Assuming our RAM's have at least $2^s + O(n)$ registers ($s \leq \lfloor k/2 \rfloor$), each with a capacity of k bits,

$$S_i^*(n,2^k) \leq S_i^*(n,2^{k-s}) + O(n), \qquad i = 1,2.$$

The leads, in turn to the following generalization of corollary 3.

Theorem 3: Assuming our RAM's have at least $2^s + O(n)$ registers ($\log n \leq s \leq \lfloor (\log R)/2 \rfloor$) each of capacity $\lceil \log R \rceil$ bits, then

$$S_i^*(n,R) = O(n\lfloor \log R \rfloor / s + n \log(s/\log n)), \qquad i = 1,2.$$

It is significant that our sorting algorithm has made effective use of a great deal more storage than time. It might be suspected that the initialization of memory would dominate the running time of our algorithm. While, in a sense, we are not obliged to defend our algorithm against this criticism (afterall, the standard RAM models [2] assume that memory has been initialized to zero, at no cost), it may be worth pointing out in this context what has

become almost a "folklore" fact (that may or may not have influenced the original RAM specifications) namely:

Fact (Initialization of RAM's): An uninitialized RAM (registers start with random values) can simulate an initialized RAM (all registers are initialized to zero) with only a constant factor of overhead.

This claim is, in essence, a restatement of problem 2.12 in [1]. The proof will, of course, be left to the reader.

5. A linear sorting algorithm for Shift-machines

The results of the preceding section provide the best upper bounds known on the complexity of integer sorting assuming logarithmic charging. As we have seen in section 3, these bounds can be substantially tightened on a RAM, if we consider uniform time complexity instead. It is natural to ask if similar improvements hold for the RAM_2 model. In this section we show the somewhat surprising result that integers from arbitrary ranges can be sorted in linear uniform time on a RAM_2. As before, exceedingly large integers play an important role in the algorithm. In contrast to our earlier algorithms, indirect addressing turns out to be unnecessary.

The following RAM_2 algorithm sorts inputs $x_1,\ldots,x_n$ in linear time.

Algorithm 3

Step 1: Read input $x_1,\ldots,x_n$.

Step 2: For $i = 1,\ldots,n$ do

$y_i \leftarrow x_i 2^n + i\, 2^{\lfloor n/2 \rfloor}$ (w.l.o.g. $n > 4$).

(This is done in order to get mutually different integers of the same order type. which satisfy: $y_i > y_j \Rightarrow y_i - y_j > 2^{\lfloor n/2 \rfloor}$.

Step 3: Compute $\omega \leftarrow \sum_{i=1}^{n} 2^{y_i}$.

(By this one gets an integer, whose binary representation has 1's exactly in the y_i-th positions, and the distance between two of these 1's is at least $2^{\lfloor n/2 \rfloor}$ bits).

Step 4: For $i = 1,\ldots,n$ do

$$a_i \leftarrow \lfloor \frac{\omega}{2^{y_i}} \rfloor 2^{y_i}.$$

(The binary representation of integers a_i has 1's only in the y_j-th position for $1 \leq j \leq n$ whereat $y_j \geq y_i$).

Step 5: Compute $b \leftarrow \Sigma a_i$.

(The binary representation of b has the following property:
In positions $y_i,\ldots,y_i + 2^{\lfloor n/2 \rfloor}$ there is a binary representation of the number of integers from $\{y_1,\ldots,y_n\}$ which are less than or equal to y_i (i.e. the rank of y_i)).

Step 6: For $i = 1,\ldots,n$ do

$$c_i \leftarrow \left\lfloor \frac{b}{2^{y_i}} \right\rfloor$$

$$d_i \leftarrow \left\lfloor \frac{c_i}{2^{\lfloor n/2 \rfloor}} \right\rfloor \cdot 2^{\lfloor n/2 \rfloor}$$

$$k_i \leftarrow c_i \dot{-} d_i$$

(At this point k_i denotes the rank of x_i in $\{x_1,\ldots,x_n\}$. That is $(k_1,\ldots,k_n)$ is a permutation of $(1,\ldots,n)$ and $k_i < k_j \Rightarrow x_i \leq x_j$. It should be clear that the sorted output could now be produced using indirect addressing. However, as the remaining steps of the algorithm show, indirect addressing can be avoided).

Step 7: 7a: $\bar{x} \leftarrow \max(x_1,\ldots,x_n)$

7b: $S_0 \leftarrow 0$

7c: For $i = 1,\ldots,n$ do

$$S_i \leftarrow S_i + x_i 2^{k_i 2^{\bar{x}}}.$$

(The binary representation of S_n is a concatenation of the integers $x_1,\ldots,x_n$ with x_i represented in bits $k_i 2^{\bar{x}},\ldots,(k_i+1)2^{\bar{x}}-1$).

Step 8: 8a: $r_0 \leftarrow S_n$

8b: For $i = 1,\ldots,n$ do

$$r_i \leftarrow \left\lfloor \frac{r_{i-1}}{2^{\bar{x}}} \right\rfloor$$

$$z_i \leftarrow r_{i-1} - r_i 2^{\bar{x}}$$

write output z_i.

(The output n-tuple $(z_1,\ldots,z_n)$ is a permutation of $(x_1,\ldots,x_n)$ satisfying $z_1 \leq z_2 \leq \cdots \leq z_n$.)

It is clear by inspection that Algorithm 3 runs in linear time assuming uniform charging. The correctness of the algorithm can be confirmed using the comments following each step. This establishes the following:

Theorem 4: $S_2(n,R) = O(n)$, for all $R \in N$.

6. Related work

6.1 Upper bounds

In addition to the basic results mentioned in the introduction, there are a number of recent results that speak directly to the problem of sorting integers in restricted ranges. P. van Emde Boas [3] was the first to describe an efficient priority queue maintenance algorithm designed specifically to deal with integers from a restricted range. In subsequent papers [4,5] van Emde Boas' result has been refined to the following:

Theorem A (van Emde Boas): There is a data structure supporting the full repertoire of priority queue operations over the range [1,R-1] that uses O(R) space and O(lg lg R) processing time per element processed. The space must be initialized in time O(R) before the structure can be used.

It follows immediately from Theorem A that $S_i(n,R) = O(n \lg \lg R) + O(R)$, the second term being a one time pre-processing (initialization) charge. While this bound itself is not competitive with conventional technique (as R increases the pre-processing cost dominates), it does point out that the incremental cost of sorting n integers in the range [0,R-1] is only O(n lg lg R).

6.2 Lower bounds

The upper bounds described above are complemented by lower bounds. For machines using real inputs, with addition, substruction, multiplication and ordinary division, but no capability of indirect addressing an $\Omega(n \log n)$ lower bound was proven by Hong Jiawei [6]. A similar lower bound is due to Paul and Simon [8] (and independently C. Rackoff [9]). They showed:

Theorem B (Paul, Simon, Rackoff): On a RAM with instruction set $\{+,\dot{-},x\}$ and capability of indirect addressing the worst case complexity of any program that sorts n numbers in a sufficiently large range is at least $\Omega(n \log n)$.

A detailed analysis of the proof given by Paul and Simon shows that for each sorting-program using only addition, substruction and multiplication there are input integers in the range $[0,n^n-1]$ requiring $\Omega(n \log n)$ steps for correct sorting.

The main idea of Paul and Simon is to associate to a given program P a binary decision tree T. Each node of the tree represents a special state of the computation of P. If $\{i_0,\ldots,i_m\}$ is the set of addresses occuring in P, the contents of these registers in each state of computation associated to a node v of T is described by a polynomial $P_{v,j}(x_1,\ldots,x_n)$ where $(x_1,\ldots,x_n)$ is the n-tuple of input-integers. Paul and Simon show that there are input integers $x_1,\ldots,x_n$ of each order type, such that different polynomials $P_{v,j}(x_1,\ldots,x_n)$ and $P_{v',j'}(x_i,\ldots,x_n)$ take different values, i.e. $x_1,\ldots,x_n$ will not set $P = \prod_{vv',jj'} (P_{v,j} - P_{v',j'})$ to zero. Because of this T has n! leaves and therefore depth at least $\frac{1}{2} n(\log n-2)$. Essential for the proof is the following:

Lemma A (Paul/Simon): If R is a polynomial in $x_1,\ldots,x_n$, deg R = m and π is a permutation of $(1,\ldots,n)$, then there is $\underline{x} \in N^n$ of order type π such that $R(\underline{x}) \neq 0$ and $(x_1,\ldots,x_n) \in \{1,\ldots,m+n\}^n$.

Employing this lemma one can see that there must be input-integers $x_1,\ldots,x_n \in [0,n^n-1]$ which will cause program P to run at least for $\frac{1}{4} n(\log n-2)$ steps.

If we cut off T at depth $\frac{1}{4} n(\log n-2)$ there are only at most $2^{\frac{1}{4} n(\log n-2)+1} \leq n^{\frac{1}{4} n}$ nodes remaining in T.

Within $\frac{1}{4}n(\log n-2)$ steps only polynomials of degree $2^{\frac{1}{4}n(\log n-2)} \leq n^{\frac{1}{4}n}$ can be produced. Therefore an appropriate input has to set a polynomial of degree

$$\binom{n^{\frac{1}{4}n}}{2} n^{\frac{1}{4}n} \leq n^{\frac{3}{4}n} - 1$$

to a value different from zero. As proven by Lemma A there exist integers $(x_1,\ldots,x_n) \in \{1,\ldots,n^{\frac{3}{4}n}\}$ fulfilling this requirement.

As shown by corollary 2 a machine with additional ability of integer-division can sort such an input in time $O(n \log \log n)$. Therefore sorting in restricted ranges is speedable by additional use of integer-division. It is an open question whether this holds in general. Furthermore it is still unknown whether a RAM_1, a machine with addition, substraction, multiplication and integer-division, can sort arbitrary n-tuples of integers in linear time.

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., The Design and Analysis of Computer Algorithm, Addison-Wesley, 1974.

2. Cook, S.A., and Reckhow, R.A., Time bounded random access machines, J.C.S.S. 7,4 (1973), 354-375.

3. van Emde Boas P., An O(n log log n) on-line algorithm for the insert-extract min problem, TR 74-221, Department of Computer Science, Cornell University, December 1974.

4. van Emde Boas, P., Kaas, R., and Zijlstra, E., Design and implementation of an efficient priority queue, Math. Systems Theory 10 (1977), 99-127.

5. van Emde Boas, P., Preserving order in a forest in less than logarithmic time and linear space, Info. Processing Letters 6,4 (June 1977), 80-82.

6. Jiawei, H., On lower bounds for time complexity of some algorithms, Scientia Sinica 32, 8/9, 890-900.

7. Keil, J.M., Computational geometry on an integer grid, M.Sc. Thesis, Department of Computer Science, University of British Columbia, April 1980.

8. Paul, W.J., and Simon, J., Decision trees and random access machines, Symposium uber Logik und Algorithmik, Zurich, 1980.

9. Rackoff, C., Private communication, 1980.

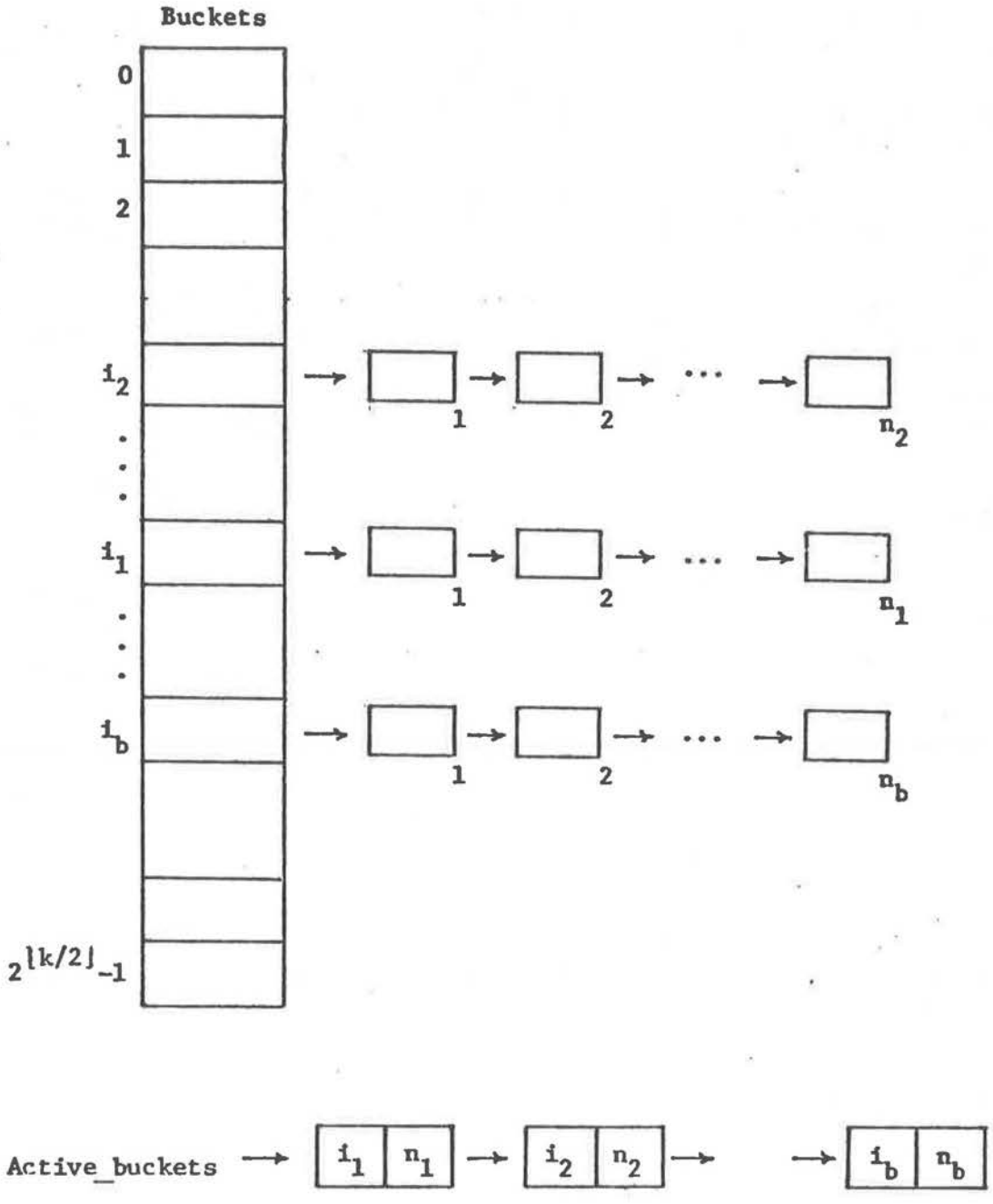
Buckets
0
1
2
i_2
i_1
i_b
$2^{\lfloor k/2 \rfloor}-1$
1
2
n_2
n_1
n_b
Active_buckets
i_1
n_1
i_2
n_2
i_b
n_b

Figure 1