BIT 25 (1985), 611–622

# NOTES ON THE COMPLEXITY OF SORTING IN ABSTRACT MACHINES

MARTTI PENTTONEN and JYRKI KATAJAINEN

*Department of Computer Science, University of Turku, SF-20500 Turku, FINLAND*

**Abstract.**

The complexity of sorting with pointer machines and successor-predecessor random access machines is studied. The size of the problem is defined as the length of the problem string. A linear time algorithm is achieved for sorting by pointer machines. For successor-predecessor random access machines linear time is sufficient in a special case.

## 1. Introduction.

It is well-known that sorting, when best done, takes $O(n \log n)$ time. In this paper we pose two questions about this statement. Which model of computation is used? How is the size $n$ of the problem defined?

We shall define the size of the sorting problem as the length of the input string, which is the concatenation of the keys separated by punctuation symbols. This gives us a uniform base for the comparison of algorithms. Sometimes in the literature the size of the problem is omitted or so vaguely defined that a direct comparison of complexity results is not possible.

In this paper we shall consider the complexity of sorting with pointer machines defined by Schönhage [8] and with random access machines. As to the pointer machines, Schönhage [8] used radix sort [5] for sorting keys with respect to equality. In fact, his technique can rather easily be extended for sorting keys of equal length with respect to their $k$-ary values in linear time, as well. We shall show that the same holds true for sorting keys of arbitrary length.

When sorting with random access machines, we restrict ourselves to the problems having a uniform key length. We shall show that the radix sort [5] will work in linear logarithmic time for "typical" key lengths, but for very long or very short keys, the time complexity approaches $O(n \log n)$. It remains open whether the linear time is possible in all cases. In fact, we do not need full arithmetic power of random access machines, only successor and predecessor operations are needed. A related work for more powerful random access

Received January 1984. Revised April 1985.

machines under a uniform cost criterion is done by Kirkpatrick and Reisch [3].

## 2. The sorting problem.

We shall consider keys that are nonempty strings over the binary alphabet. The binary values of the keys induce a linear order among them, thus for instance $1101 < 10011$, because $13 < 19$.

We shall use the symbol # as the separator of the keys. The *sorting problem* is to compute the function mapping any input string

$$w_0 \# w_1 \# \ldots \# w_{t-1} \#\#$$

where the keys $w_i$ are binary strings, to the string

$$w_{p(0)} \# w_{p(1)} \# \ldots \# w_{p(t-1)} \# \#$$

where $p$ is a permutation of $\{0, 1, \ldots, t-1\}$ such that $w_{p(i)} \leqslant w_{p(i+1)}$ $(i = 0, 1, \ldots, t-2)$ in the linear order of binary numbers. The **size** of the sorting problem is the length of the input string (or the number of letters 0, 1, # in it).

Thus, for example, the size of 110 # 10 # 01 # 1001 ## is 16, and when sorted it becomes 01 # 10 # 110 # 1001 ##. One should note that we sort according to the binary value and not according to the lexicographic order. In the latter case the result would be 01 # 10 # 1001 # 110 ##.

## 3. Sorting in linear time with pointer machines.

In this section we shall show that a pointer machine sorts any problem of size $n$ in time $O(n)$. Of course, this does not contradict the well-known [1] lower bound that sorting $t$ keys requires $\Omega(t \log t)$ comparisons of keys.

The definition of the pointer machine was proposed already by Kolmogorov and Uspenskij [6] and by Knuth [4, Section 2.6.]. A fundamental study was done by Schönhage [8]. Tarjan [9] also has used this kind of machine.

A *pointer machine* (PM) has an *input tape* which is scanned by a one-way read-only head, and an *output tape* with one-way write-only head. We assume that input and output alphabets contain only letters 0, 1, and #. The memory of a PM is a graphlike structure, bearing resemblance with a dynamic record structure which is available in many programming languages. A memory *cell* consists of $k$ *pointers* labelled with the letters from a *pointer alphabet* $P$, where $k$ is a fixed number, $k \geqslant 2$. There are $k$ letters in the pointer alphabet, and different pointers of a cell are labelled with different letters. Thus, there is a one-to-one correspondence between pointers and labels. The memory structure can be considered as a digraph whose edges are labelled and which has a constant fan out $k$.

For the access to the cells of the memory, there is a special centre pointer which points to a cell called the *centre* of the memory. All nodes are accessed via the centre. The computation starts with a centre whose pointers are all

idempotent, i.e. they point from the centre to itself. If $p_1, p_2, \ldots, p_j$ are pointer labels, we denote by $n(p_1p_2\ldots p_j)$ the cell which is accessed from the centre by following the pointers $p_1, p_2, \ldots, p_j$. Especially, $n(e)$ refers to the centre. Whenever a new cell is created, the pointer **new** is set to it, and its own pointers are set to itself.

A pointer machine has a fixed program which is a sequence of labelled or unlabelled *instructions*, separated by semicolons. A *label* is a letter from a program label alphabet and it is separated from the following instruction by a colon. The set of instructions that are available is the following:

| **instruction** | **meaning** |
|---|---|
| **input** $l_0, l_1, l$ | causes the next input symbol a $\in \{0, 1, \#\}$ to be read and the control transferred to the instruction with the label $l_0, l_1$ or $l_\#$, respectively, |
| **output** $b$ | outputs the symbol $b$, $b \in \{0, 1, \#\}$, |
| **goto** $l$ | transfers the control to the instruction labelled with $l$, |
| **halt** | ends the computation, |
| **create new cell** | creates a new cell with idempotent pointers and sets the pointer **new** to it, |
| **move centre to** $n(u)$ | where $u \in P^* \cup \{\textbf{new}\}$, moves the centre to the cell $n(u)$, |
| **set** $p$ **from** $n(u)$ **to** $n(v)$ | where $u, v \in P^* \cup \{\textbf{new}\}$, sets the $p$-pointer from $n(u)$ to $n(v)$, |
| **if** $n(u) = n(v)$ | **then** $I$ causes the instruction $I$ of previous types to be executed if the cells $n(u)$ and $n(v)$ are same. |

The set of instructions is given in order to state exactly, what are the elementary steps of the PM. However, for the sake of readability we use **if-then-else**, **while-do**, and **repeat-until** control structures which can easily be translated into the basic formalism. Additionally, we assume that a cell can be labelled by a *letter* from a *memory alphabet*. The letter can easily be replaced by an additional pointer. Indeed, this technique is explained in [8]. We denote by $c(p_1p_2\ldots p_j)$ the letter contained in the cell $n(p_1p_2\ldots p_j)$. In the following the pointers not explicitly mentioned are all idempotent, i.e. they point to the cell itself. We do not even draw idempotent pointers in a figure. The pointer alphabet is hereafter assumed to be $\{B, E, K, L, S\}$.

The *time complexity* of a PM program is simply the number of instructions executed.

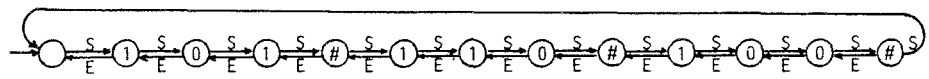


Fig. 1. An $S$-chain.

EXAMPLE 3.1. *The following small program constructs an S-chain corresponding to a sorting problem. For input* 101 # 110 # 100 ## *the results is seen in Figure* 1.

**repeat**
  **create new cell**
  **set** $S$ **from** $n(E)$ **to** $n$(**new**)
  **set** $E$ **from** $n(ES)$ **to** $n(E)$
  **set** $E$ **from** $n(e)$ **to** $n(ES)$
  $c(E) \leftarrow$ **input**
**until** $c(E) = \#$ **and** $c(EE) = \#$
**set** $S$ **from** $n(E)$ **to** $n(e)$
**set** $E$ **from** $n(e)$ **to** $n(e)$.

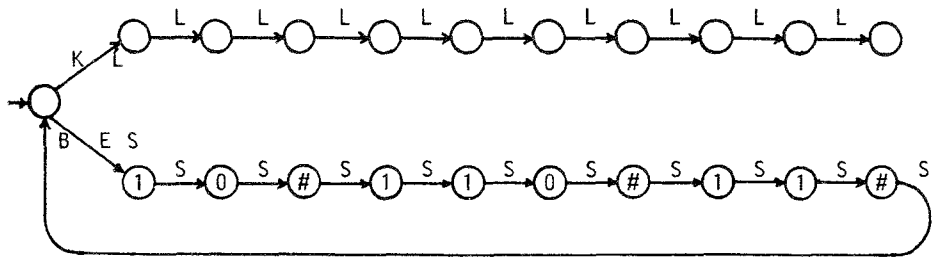


Fig. 2. Memory structure before length sorting.

Sorting by a PM is done in two stages. During the first stage the key words $w_i$ of the input string $w_0 \# w_1 \# \ldots \# w_{t-1} \#\#$ are read and distributed to lists of words of equal length, when initial zeroes are not counted. Consequently, in the first stage the keys are sorted on a very coarse level. We shall see that this can be done in linear time. To complete the sorting, the lists formed during the first stage are stored. This is done in the second stage in linear time using radix list sorting [5]. The final output is the concatenation of the sorted sublists.

Our definition of the sorting problem requires that the leading zeros of the keys must not be omitted while sorting. This requirement causes some minor inconveniencies in the constructions, as for example 0010, 11, and 010 are to be considered as having equal length two.

LEMMA 3.1. *There is a PM which sorts the problem* $w_0 \# w_1 \ldots \# w_{t-1} \#\#$ *of size n in time* $O(n)$ *into the increasing order of the lengths of* $w_i$, *when the leading zeros are not counted.*

PROOF. Denote $w = w_0 \# w_1 \ldots \# w_{t-1} \#\# = a_1 a_2 \ldots a_n \quad (a_i \in \{0, 1, \#\})$. While reading the input string, two chains of cells are constructed. The cells in the $S$-chain contain the input letters $a_1, a_2, \ldots, a_n$ in the original order. The other chain is constructed with $L$-pointers, and it will be used for measuring the length of a key. In the $L$-chain, the $S$-pointer of a cell is used to collect the keys

of that length. Those $S$-pointers are initially idempotent meaning that the corresponding lists are empty. Also two additional pointers $B$ (for beginning) and $E$ (for end) are set from the centre to $n(S)$; they will mark the beginning and the end of the active key. Finally, the pointer $K$ is set from the centre to $n(L)$ corresponding to the list of keys of the length 1. For input 10 # 110 # 11 ##, for example, the configuration is shown in Figure 2.

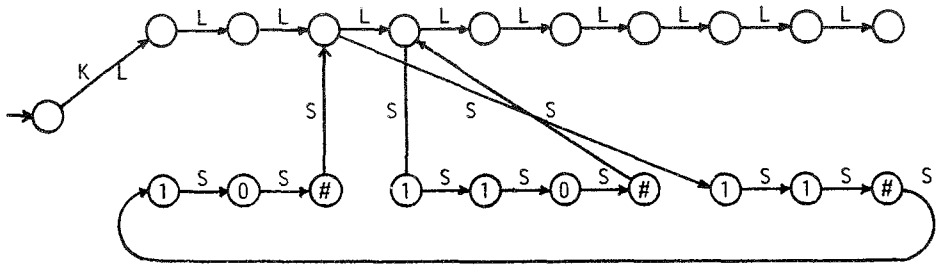


Fig. 3. Memory structure after length sorting.

Then the keys are distributed according to their lengths using the program

```
while n(B) ≠ n(e) do
    while c(E) = 0 do                    {pass leading zeroes}
        set E from n(e) to n(ES)
    end while
    while c(E) ≠ # do                    {measure the length of the keys}
        set E from n(e) to  n(ES)
        set K from n(e) to n(KL)
    end while
    set S from n(e) to n(ES)             {remember the next key if any}
    set S from n(E) to n(KS)             {insert the key at the beginning...}
    set S from n(K) to n(B)              {...of key list of the same length}
    set B from n(e) to n(S)              {initializations for the next key}
    set E from n(e) to n(S)
    set K from n(e) to n(L)
end while
```

The principal **while**-loop is executed as many times there are keys in $w$. The first of the two inner **while**-loops passes the initial zeros of the key. In the second inner **while**-loop, the end of the key is recognized. The remaining statements locate the current key to the beginning of the list corresponding to its length and initialize the temporary pointers ready for the next key. In the case of our example, the program yields Figure 3.

A list of keys in the increasing order of lengths can now be output easily, but in our case it is not useful, because in the next lemma the lists of keys of equal length are sorted according to their binary values. It is obvious that only linear time has been used.

LEMMA 3.2. *A problem* $w = w_0 \# \ldots \# w_{t-1} \#\#$ *of size* $n$, *where all keys,* $w_i$ *have equal number of significant bits, can be sorted in* $O(n)$ *time by a PM.*

PROOF. Let us assume that the problem $w$ is stored in the memory in an $S$-chain. Before sorting cycles, the $S$-chain must be prepared for sorting. For each key $w_i$, an $E$-pointer is set from the first letter of $w_i$ to the terminator $\#$ of $w_i$. For all keys, a $B$-chain is set from the terminator via all bits to the first bit, and from the first bit to the last bit. A $B$-pointer is also set from the centre to the first bit of the first key. In addition, two new cells are created. The cell $n(K)$ will collect the keys with zero in certain position, and the cell $n(L)$ will collect the keys with one, respectively. The $E$-pointers of the cells $n(K)$ and $n(L)$ will indicate the end of the key lists. Initially the lists are empty and thus those $E$-pointers are idempotent. For example, the configuration for $101 \# 110 \# 100 \#$ is as shown in Figure 4.

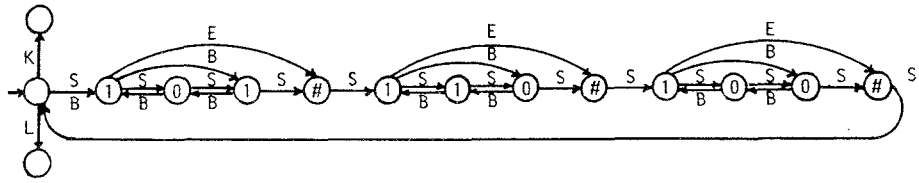


Fig. 4. Memory structure before the first distributing cycle.

In the first cycle, keys ending with 0 are joined to the end of $K$-list, and the keys ending with 1 to the end of $L$-list. When all keys have been added to their respective lists, the $K$-list and the $L$-list are concatenated, and the cycle for the last but one bit begins, and so forth. This is formalized in the following program:

```
set E from n(e) to n(e)
repeat
  while n(B) ≠ n(e) do                    {distribute all keys}
    if c(BB) = 0 then
      set S from n(KE) to n(B)            {join key to the end of K-list}
      set E from n(K) to n(BE)
    else
      set S from n(LE) to n(B)            {as above, for L-list}
      set E from n(L) to n(BE)
    end if
    if n(BB) ≠ n(B) then                  {take a new bit if not all bits used}
      set B from n(B) to n(BBB)
    else
      set E from n(e) to n(L)
    end if
```

```
    set B from n(e) to n(BES)
  end while
  set S from n(LE) to n(e)          {concatenate K-list and L-list}
  set S from n(KE) to n(LS)
  set S from n(e) to n(KS)
  set B from n(e) to n(S)           {initializations for the next cycle}
  set E from n(K) to n(K)
  set E from n(L) to n(L)
until n(E) ≠ n(e)
```

Obviously, the inner loop is executed $t$ times, and the outer loop as many times as there are bits in the shortest of the keys, or equivalently as many times as there are significant bits in the keys. Consequently, the total time is linear.

Combining the results of Lemmas 3.1 and 3.2 we get

THEOREM 3.1. *The general sorting problem* $w_0 \# \ldots \# w_{t-1} \#\#$ *of size n can be performed in* $O(n)$ *time by a pointer machine.*

This result, of course, is optimal as far as the constant coefficient of $O(n)$ is not taken into account.

NOTE 3.1. Also lexicographic sorting in linear time by PMs is possible. In fact, it is easier than $k$-ary sorting, since trie sorting [5] is directly applicable to PMs.

## 4. Sorting with random access machines.

The random access machine (RAM) has been proposed as an abstract model of existing computers. Indeed, the random access machine captures essential features of the machine languages. In order to achieve generality and simplicity, several idealizations have been done, for example the memory locations of RAM can contain arbitrarily large numbers.

Our definition of the RAM is not different from those found in the literature [1, 2, 8], but we have to define it more exactly. In our machine, the memory locations are considered as bit strings rather than integers. The input and output of bit strings is rather problematic. We decided to use a buffer for that purpose. Also we do not need full arithmetics, successor and predecessor operations are sufficient for our purposes.

The *random access machine* (RAM) is an abstract machine which consists of a fixed *program*, an *input* tape and an *output* tape containing words over $\{0, 1, \#\}$, and a sequence BF, AC, $0, 1, 2, \ldots$ of *memory locations* containing bit strings of arbitrary length, sometimes interpreted as integers. BF is called the *input/output* buffer, through which all input and output is passed. AC is called the *accumulator*, in which all computation is done. The bit string contained in the memory location $m$ is denoted with $c(m)$. The program of a RAM is a finite

sequence of *labelled* or *unlabelled instructions*. A label is a positive integer, and an instruction is one of the following:

| **instruction** | **meaning** |
|---|---|
| **read** $l_0, l_1, l_\#$ | 0,1 or nothing is attached to the end of $c(BF)$ and the control is transferred to the instruction labelled with $l_0$, $l_1$, or $l_\#$, respectively, |
| **write** | if $c(BF)$ is nonempty, then the first bit is output and removed from $BF$, else # is output, |
| **reset** | the input head is reset to the beginning of the input tape for rereading, |
| **halt** | the machine halts the computation, |
| **jump** $l$ | jump to the instruction labelled with $l$, |
| **jumpp** $l$ | if $c(AC) > 0$ then jump to the instruction labelled with $l$, else to the next instruction, |
| **loadim** $n$ | $c(AC) \leftarrow n$, $n$ is an integer, |
| **load** $m$ | $c(AC) \leftarrow c(m)$, $m$ is a memory location, |
| **loadid** $m$ | $c(AC) \leftarrow c(c(m))$, |
| **store** $m$ | $c(m) \leftarrow c(AC)$, |
| **storeid** $m$ | $c(c(m)) \leftarrow c(AC)$, |
| **succ** | $c(AC) \leftarrow c(AC)+1$, |
| **pred** | $c(AC) \leftarrow c(AC)-1$. |

At the beginning of the computation all memory locations are assumed to contain the empty string or zero.

We shall use *logarithmic cost* as defined for example in [1]. Whenever data are handled, the size of the contents and the address must be taken into account. The following small lemma is needed in our main theorem, but it may also have some interest of its own.

LEMMA 4.1. *For any problem* $w_0 \# w_1 \# \ldots w_{t-1} \#\#$ *of size* $n$, $\log t$ *and* $\log\log t$ *can be computed in* $O(n)$ *time.*

PROOF. For the counting of the keys, $4 \log t + 4$ memory locations are used, four locations for each bit of the binary representation of $t$. At a moment when $d$ keys have been counted, the locations 6, 10, 14,... contain the bits of the binary representation of $d$, in increasing order of significance. The locations 4, 8, 12,... contain 0, except the last one preceding the highest non-zero bit of $\log d$, which contains 1. The locations 0, 1, 2, 3 are used for auxiliary computations, and the locations 5, 7, 9, 11,... are used for the computation of $\log\log t$.

The keys are counted as follows. Each time a key is encountered on the input tape, the key count whose bits are in the locations 6, 10, 14,... is increased by 1. This is done by successively scanning these locations via a pointer stored in

location 3. If a new bit is needed (when key count $d$ reaches a power of 2), the end marker 1 in one of the locations 4, 8, 12,... is moved four locations to the right. Hence, when all keys have been counted, the end marker is at the location $4 \log t$.

Analogously, the number of zeros in the locations 4, 8, 12,... is counted with locations 7, 11, 15,... until the end marker 1 is encountered. Here too, a 1 in one of the locations 5, 9, 13,... shows the position of the highest non-zero bit. Now $\log \log t$ can be computed into the location 0 by counting the zeros in locations 5, 9, 13,... preceding the end marker 1.

Obviously, the first stage is more time consuming than the second one, and therefore we can restrict ourselves to the analysis of the first case. Of the $t$ counting steps, $t/2$ reach only the lowest bit, $t/4$ reach the second bit, $t/8$ the third bit and so forth. Hence, the total cost of these steps is

$$(t/2) \cdot 1 \cdot 1 + (t/4) \cdot 2 \cdot \log 2 + (t/8) \cdot 3 \cdot \log 3 + \ldots + 1 \cdot \log t \cdot \log \log t = O(t)$$

as is easily seen by using the quotient test. Hence, the scanning of the input taking $O(n)$ time is decisive in the time complexity.

It follows from theorem 4.1 and a general simulation theorem of Schönhage [8] that any problem can be sorted in $O(n \log n)$ time by a successor random access machine. Now we begin to look for a better algorithm. We restrict ourselves to problems with uniform key length.

Our algorithm will use $k$-ary radix sorting [5]. The crucial idea is to choose $k$ so large that it balances with the number of keys. Indeed, $k = t$ proves to be a good choice. This implies that the keys are not handled as bit strings but as block strings, each block consisting of $\log t$ bits. In fact, the proof is rather similar to the proof of lemma 3.2, but here we need blocking in order to cover the cost of pointers.

LEMMA 4.2. *Let $w_0 \# w_1 \# \ldots \# w_{t-1} \#\#$ be a sorting problem of $t$, $t > 1$, keys of equal length, and let $b = \lceil |w_i| / \lceil \log t \rceil \rceil$. Then there is a successor-predecessor RAM that sorts the problem in time $O(tb \log (tb))$.*

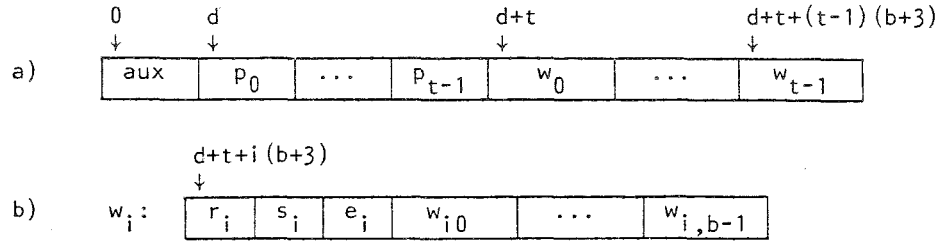


Fig. 5. a) Memory map of the RAM. b) A key in the map.

PROOF. By lemma 4.1, $\log\log t$ can be computed in $O(n)$ time. By storing 1 in location $\log\log t$, $\log t$ can be represented as a bit vector, and by the techniques of lemma 4.1 $\log t$ bits of the input can be counted (down) and loaded into the input buffer in $O(\log t)$ time. A certain memory location, say location 0, is used for pointing where this block is stored. As there are $tb$ blocks, the cost of storing addresses is $\log tb$. Hence, the configuration corresponding to the memory map of Figure 5 can be constructed in $O(tb \log (tb))$ time.

In the map, there are $d$ auxiliary pointers and counters, where $d$ is a small constant. The next memory locations correspond to the buckets of the radix sort; $p_i$ shows the place of the first key whose active key has $t$-ary value $i$, if any. At the beginning, each $p_i$ is empty pointer 0. The last $t(b+3)$ memory locations contain the keys, each divided in $b$ blocks with three auxiliary pointers. The first of these pointers, say $r_i$, points to the active block of $w_i$, initially to the last one, i.e. $r_i = d+t+(i+1)(b+3)-1$. The second pointer $s_i$ gives the successor of this key at this stage of the sort, originally $s_i = d+t+(i+1)(b+3)$ for $i = 0,\ldots,t-2$ and $s_{t-1} = 0$. The third pointer $e_i$ shows where the $s$-chain beginning at $w_i$ ends, if $w_i$ happens to begin an $s$-chain.

The sorting consists of $b$ distributing stages and $b$ collecting stages which alternate. During a distributing stage, each key is added to the end (pointed by the $e$-pointer) of the $s$-chain corresponding to its active block (accessed by the $r$-pointer). Assume that the active block of $w_i$ be $j$, i.e. the $r$-pointer of $w_i$ has value $c(c(d+t+i(b+3))) = j$. If $c(d+j) = 0$, then the $s$-chain corresponding to $j$ is empty. In this case $w_i$ is made the first and only element of the $s$-chain of $j$ by the assignments

| | |
|---|---|
| $c(d+j) \leftarrow d+t+i(b+3)$ | $\{p_j$ points to $w_i\}$ |
| $c(d+t+i(b+3)+2) \leftarrow d+t+i(b+3)$ | $\{w_i$ becomes the tail of the $s$-chain$\}$ |
| $c(d+t+i(b+3)+1) \leftarrow 0$ | $\{w_i$ has no successor$\}$ |
| $c(d+t+i(b+3)) \leftarrow c(d+t+i(b+3))-1$ | {next active block of $w_i$} |

Note that although the assignments seem to contain addition and multiplication, in fact only indirect addressing, successor and predecessor functions are needed.

If $p_j = c(d+j) > 0$, there are already keys in the $j$-bucket. In this case $w_i$ is joined to the end of the chain as follows.

| | |
|---|---|
| $c(c(c(d+j)+2)+1) \leftarrow d+t+i(b+3)$ | $\{w_i$ is joined to the end of $s$-chain$\}$ |
| $c(c(d+j)+2) \leftarrow d+t+i(b+3)$ | $\{w_j$ becomes the end of $s$-chain$\}$ |
| $c(d+t+i(b+3)+1) \leftarrow 0$ | $\{w_i$ has no successor$\}$ |
| $c(d+t+i(b+3)) \leftarrow c(d+t+i(b+3))-1$ | {next active block of $w_i$} |

The complexity of these assignments is $O(\log (tb))$. In this way, following the $s$-chain of the previous stage, all keys are distributed to the chains corresponding

to their active blocks, and this operation has taken $O(t \log (tb))$ time altogether.

When a distributing stage has been completed, a collecting stage begins. Every nonempty $s$-chain beginning at $p_i$ $(i = 0,\ldots,t-1)$ is appended to the new $s$-chain in turn, and $p$-pointers are set to 0. The collection is controlled by a pointer to the bucket area and another pointer to the key area. When a bucket is empty, it is passed, and control moves to the next $p$-pointer. When a nonempty bucket is encountered, it is appended to the new chain and the key pointer of the new $s$-chain is set to the end of this bucket by the $e$-pointer of the first member of this bucket. The total cost of handling $p$-pointers is $O(t \log t)$, and the total cost of moving the key pointer is $O(t \log (tb))$. Hence a collecting stage can be completed in $O(t \log (tb))$ time.

We have proved that both distributing and collecting stages take $O(t \log (tb))$ time each. As there are $b$ blocks in a key, the total time of these stages is $O(tb \log (tb))$. Outputting, evidently, can be done is $O(tb \log (tb))$ time with the same technique as inputting. From this lemma we obtain immediately:

THEOREM 4.1. *If all keys of the sorting problem* $w_0 \# w_1 \# \ldots \# w_{t-1} \#\#$ *of size* $n$ *have the same length* $k$ *satisfying* $\log n \leqslant k \leqslant n^{1-\varepsilon}$ *for some positive constant* $\varepsilon$, *then the problem can be sorted in linear time by a random access machine.*

PROOF. By lemma 4.2 and the first inequality, the problem can be sorted in time $O(tb \log (tb))$, where $b = k/\log t$. By the second inequality, $\log (tb) = \log (tk/\log t) = O(\log t)$. Hence, $O(tb \log (tb)) = O(tk) = O(n)$.

If the key length decreases to 1 or grows to $n$, the time bound of lemma 4.2 approaches $O(n \log n)$. In these extremal cases some other algorithms give better bounds. For very short keys, the trivial algorithm outputting all zeros at the first reading, all ones at the second reading and so forth, gives a better bound. As it is well-known [1], mergesort needs $O(t \log t)$ comparisons of keys, and one can see that a successor-predecessor RAM is sufficient for programming it. If the key length $n/t$ is taken into account, its time complexity with respect to the length of the input is $O(t \log t(n/t)) = O(n \log t)$, which is better than ours for large key lengths.

## 5. Discussion and open problems.

We gave a linear time sorting algorithm for pointer machines which, of course, is optimal. However, in the domain of sorting with other machines there remain many open problems.

The multitape Turing machine is the traditional model of computation for complexity studies. Using the polyphase merge method [4] one can show [7] that a sorting problem of $t$ keys and size $n$ can be solved in $O(n \log t)$ time with a Turing machine having at least three tapes. For two-tape Turing machines $O(n \log n \log t)$ is achieved and for one-tape machines $O(nt)$. It would be

interesting to know if these time bounds are optimal for Turing machines.

We have not studied lexicographic sorting by successor-predecessor RAMs. Additionally, we have not been able to achieve a good time bound for the general problem of sorting keys of variable length according to their $k$-ary values. It can be shown that if the length of all keys is at least logarithmic then the sorting according to length can be done in linear time. Unfortunately, unlike the case of PMs, this does not imply a linear time algorithm even for this case.

Another interesting question is, which assumptions about our RAM model can be weakened. We do not see any easy way to remove the predecessor or reset instructions or input/output buffer without increasing the time bound. The usefulness of these properties is seen by the following simple test problems. How fast can a RAM output two copies of an input string without using a buffer or a reset? How fast can a RAM output the reversal of an input string with or without the predecessor instruction? These functions can easily be computed in linear time by Turing machines, but there is no obvious way to compute them in linear time by RAMs without the abovementioned facilities.

**Acknowledgements.**

We want to thank Jukka Teuhola for the note 3.1 on trie sorting. We are grateful to Arnold Schönhage for his criticism on an earlier version of this paper, and especially for the lemma 4.1.

## REFERENCES

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. S. A. Cook and R. A. Reckhow, *Time bounded random access machines*, Journal of Computer and System Sciences 7 (1973), 354–375.
3. D. Kirkpatrick and S. Reisch, *Upper bounds for sorting integers on random access machines.* Theoretical Computer Science 28 (1984), 263–276.
4. D. Knuth, *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison-Wesley 1968.
5. D. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching.* Addison-Wesley, 1975.
6. A. N. Kolmogorov and V. A. Uspenskij, *K opredeleniju algoritma.* Uspehi matematičeskih nauk XIII (1958), No 4, 3–28. English translation Amer. Math. Soc. Transl. 29 (1963), 217–245.
7. M. Penttonen and J. Katajainen, *Notes on the complexity of sorting in abstract machines.* Report A36, Department of Computer Science, University of Turku, 1983.
8. A. Schönhage, *Storage modification machines.* SIAM Journal on Computing 9 (1980), 490–508.
9. R. E. Tarjan, *A class of algorithms which require nonlinear time to maintain disjoint sets.* Journal of Computer and System Sciences 18 (1979), 100–127.