A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms

Lars Arge* Mikael Knudsen†
Kirsten Larsent‡

Aarhus University, Computer Science Department
Ny Munkegade, DK-8000 Aarhus C.

August 13, 1992

August 13, 1992

Abstract

We show a relationship between the number of comparisons and the number of I/O-operations needed to solve a given problem. We work in a model, where the permitted operations are I/O-operations and comparisons of two records in internal memory. An I/O-operation swaps B records between external memory and the internal memory (capable of holding M records). An algorithm for this model is called an I/O-algorithm. The main result of this paper is the following: Given an I/O-algorithm that solves an n-record problem P_n, using $I/O(\overline{x})$ I/O's on the input $\overline{x}$, there exists an ordinary comparison algorithm that uses no more than $n \log B + I/O(\overline{x}) \cdot T_{\text{merge}}(M - B, B)$

*E-mail: large@daimi.aau.dk
†E-mail: kmnk@daimi.aau.dk
‡E-mail: kiki@daimi.aau.dk

comparisons on input $\overline{x}$ $T_{\text{merge}}(n, m)$ denotes the number of comparisons needed to merge two sorted lists of size n and m, respectively. We use the result to show lower bounds on the number of I/O-operations needed to solve the problems of sorting, removing duplicates from a multiset and determining the mode (the most frequently occurring element in a multiset). Aggarwal and Vitter have shown that the sorting bound is tight. We show the same result for the two other problems, by providing optimal algorithms.

Topics: Algorithms and Data Structures, Computational Complexity.

1 Introduction

In the studies of complexity of algorithms, most attention has been given to bounding the number of primitive operations (for example comparisons) needed to solve a given problem. However, when working with data materials so large that they will not fit into internal memory, the amount of time needed to transfer data between the internal memory and, say, the hard disc is not neglectable. Even with the rapid development in storage devices, it should be safe to say that primitive operations in the internal memory can be carried out within microseconds, whereas transfers between internal memory and the hard disc cost in the order of milliseconds.

Aggarwal and Vitter [1] have considered the I/O-complexity of sorting, fast Fourier transformation, permutation networks, permuting and matrix transposition. They give asymptotically matching lower and upper bounds for these problems. The basic idea in their proofs is to count how many permutations can be generated with a given number of I/O-operations and to compare this to the number of permutations needed to solve a problem. They do not in general make any restrictions on the operations allowed in internal memory, except that records are considered to be atomic and cannot be divided into smaller pieces. Only when the internal memory is extremely small, the comparison model is assumed.

We will be using the same model of computation except that in general we

will limit the permitted operations in the internal memory to comparisons. Our result provides a lower bound for any problem that fits into this model, among these is sorting where we obtain the same lower bound as in [1].

We shall consider n-record problems, where in any start configuration the n records - $x_1, x_2, \ldots, x_n$ - reside in secondary storage. The number of records that can fit into internal memory is denoted M and the number of records transferable between internal memory and secondary storage in a single block is denoted $B(1 \leq B \leq M \leq n)$. The internal memory and the secondary storage device together are viewed as an extended memory with at least $M + n$ locations. The first M locations in the extended memory constitute the internal memory - we denote these $s[1], s[2], \ldots, s[M]$ - and the rest of the extended memory constitute secondary storage. The k'th *track* is defined as the B contiguous locations $s[M+(k-1)B+1], s[M+(k-1)B+2], \ldots, s[M+kB]$ in extended memory, $k = 1, 2, \ldots$. An I/O-operation is now an exchange of B records between the internal memory and a track in secondary storage.

In the next section, we define I/O-trees on which the main result in section 3 is based. In section 4, we discuss the generality of the I/O-tree model, and in section 5, we give optimal algorithms for two problems concerning multisets, namely determining the mode and removing duplicates.

2 Definition of I/O-trees

An I/O-tree is a tree with two types of nodes: comparison nodes and I/O-nodes. Comparison nodes compare two records x_i and x_j in the *internal memory* using $<$ or $\leq$. A comparison node has two outgoing edges, corresponding to the two possible results of the comparison. An I/O-node performs an I/O-operation, that is, it swaps B (possibly empty) records in the internal memory with B (possibly empty) records from secondary storage. The B records from secondary storage must constitute a track (see figure 1).

To each I/O-node, we attach a predicate Q and two functions π and π'. The predicate Q contains information about the relationship between the x_i's. We define the predicate recursively: First we attach a predicate P_k to each edge from a comparison node k. If the node made the comparison $x_i < x_j$

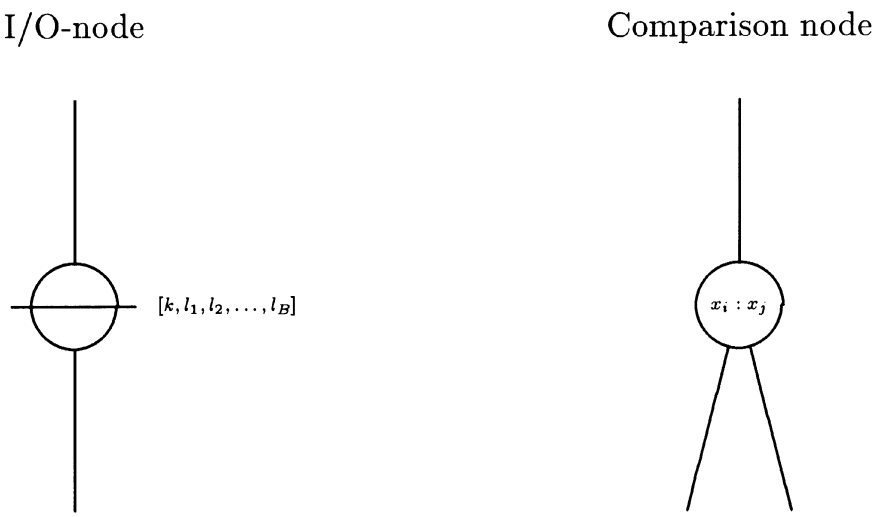

Figure 1: Node-types: An *I/O-node* swaps the B records $s(l_1), \ldots, s(l_B)$ with the B records in the k'th track, as denoted by the I/O-vector $[k, l_1, l_2, \ldots, l_B]$, where $l_1, l_2, \ldots, l_B \in \{1, \ldots M\}$ and are pairwise different, and $k \in \{1, 2, \ldots\}$. A *comparison node* compares x_i with x_j. x_i and x_j must be in internal memory.

the predicate $x_i < x_j$ is attached to the left edge, and $x_i \geq x_j$ the right edge. Similarly with $\leq$. We now consider a path S where we number the I/O-nodes $s_0, s_1, s_2, \ldots$ starting in the root and ending in the leaf.

Q_{s_i} is then defined as follows: $Q_{s_0} = True$
$Q_{s_i} = Q_{s_{i-1}} \wedge P_1 \wedge P_2 \wedge \ldots \wedge P_l$

where $P_1, P_2 \ldots P_l$ are the predicates along the path from s_{i-1} to s_i (see figure 2).

The π's contain information about where in the extended storage the original n records - $x_1, x_2, \ldots, x_n$ - are placed. More formally, we have: $\pi : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots\}$, where $\pi(i) = j$ means that x_i is in jth cell in the extended memory. Note that π is one-to-one. π' is the result of performing an I/O-operation in a configuration described by π, i.e., a track, consisting of B records, is swapped with B records from the internal memory (as denoted

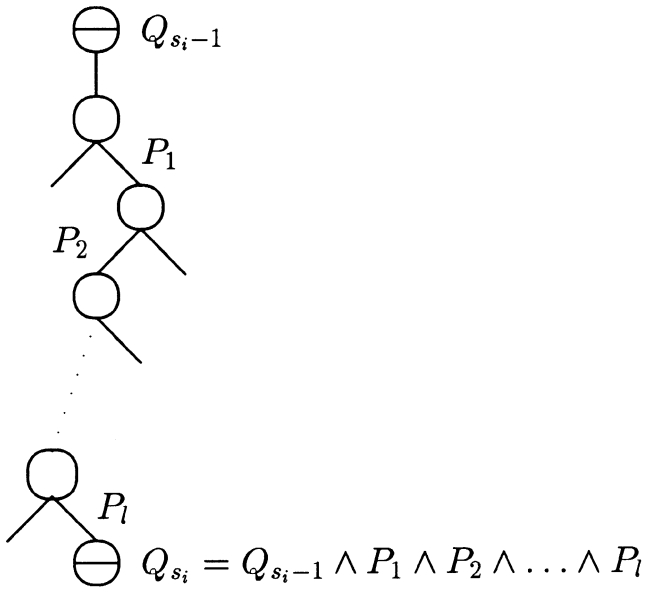

Figure 2: The predicate Q_{s_i} is defined recursively from the predicate $Q_{s_{i-1}}$ and the predicates along the path between the two I/O-nodes.

by the $(B+1)$-vector in figure 1). More formally, $\pi' = \pi$ except for the following:

$$\begin{array}{lcl} \pi'(\pi^{-1}(l_1)) & = & M + (k-1)B + 1 \\ \pi'(\pi^{-1}(M + (k-1)B + 1)) & = & l_1 \\ \vdots & & \\ \pi'(\pi^{-1}(l_B)) & = & M + kB \\ \pi'(\pi^{-1}(M + kB)) & = & l_B \end{array}$$

π in an I/O-node is, of course, equal to π' in its closest ancestor, i.e., $\pi_{s_i} = \pi_{s_{i-1}}$.

Definition 1 *An I/O-tree is a tree consisting of comparison and I/O-nodes. The root of the tree is an I/O-node where $\pi_{root}(i) = M+i$, i.e. corresponding to a configuration where there are n records residing first in secondary storage and the internal memory is empty. The leaves of the tree are I/O-nodes, again corresponding to a configuration where the n records reside first in secondary storage (possibly permuted with respect to the start configuration)*

and the internal memory is empty. This means that $\pi'_{leaf}(i) \in \{M+1, M+2 \ldots M+n\}$ *for all* i.

Definition 2 *If* T *is an I/O-tree then* $\text{path}_T(\overline{x})$ *denotes the path* $\overline{x}$ *follows in* T. $|\text{path}_T(\overline{x})|$ *is the number of nodes on this path.*

We split the problems solvable by I/O-trees into two classes: decision problems and construction problems. Decision problems are problems where we, given a predicate Q_P and a vector $\overline{x}$, want to decide whether or not $\overline{x}$ satisfies Q_P. Construction problems are problems where we are given a predicate Q_P and a vector $\overline{x}$, and want to make a permutation ρ, such that $\rho(\overline{x})$ satisfies Q_P.

Definition 3 *An I/O-tree* T *solves a derision problem* P, *if the following holds for every leaf* l:

$$\begin{array}{lcrl} (\forall \overline{x} : Q_l(\overline{x}) & \Rightarrow & Q_P(\overline{x})) & \vee \\ (\forall \overline{x} : Q_l(\overline{x}) & \Rightarrow & \neg Q_P(\overline{x})) & \end{array}$$

An I/O-tree T *solves a construction problem* P, *if the following holds for every leaf* l:

$$\forall \overline{x} : Q_l(\overline{x}) \Rightarrow Q_P(x_{\pi'^{-1}_l(M+1)}, x_{\pi'^{-1}_l(M+2)}, \ldots, x_{\pi'^{-1}_l(M+n)})$$

It is important to note that an I/O-tree reduces to an ordinary comparison tree solving the same problem, if the I/O-nodes are removed. This is due to the fact that the comparison nodes refer to records (numbered with respect to the initial configuration) and not to storage locations.

3 The Main Result

Theorem 1 *Let* P_n *be an* n*-record problem,* T *be an I/O-tree solving* P_n *and let* $I/O_T(\overline{x})$ *denote the number of I/O-nodes in* $\text{path}_T(\overline{x})$. *There exists an ordinary comparison tree* T_c *solving* P_n *such that the following holds:*

$$|\text{path}_{T_c}(\overline{x}) \leq n \log B + I/O_T(\overline{x}) \cdot T_{\text{merge}}(M-B, B)$$

where $T_{\text{merge}}(n, m)$ *denotes the number of comparisons needed to merge two sorted lists of length* n *and* m, *respectively.*

Proof We will prove the theorem by constructing the comparison tree T_c, but first we want to construct another I/O-tree T' that solves P_n from the I/O-tree T.

We consider a *comparison subtree* of T - an inner comparison tree of T with an I/O-node as the root and its immediately succeeding I/O-nodes as the leaves (see figure 3). A characteristic of this tree is that, except from the I/O-nodes in the root and in the leaves, it only contains comparison nodes that compare records in the internal memory, i.e. comparisons of the form $x_i < x_j (x_i \leq x_j)$ where $\pi(i), \pi(j) \in \{1, .., M\}$. In other words $Q_{i_1}, Q_{i_2}, \ldots, Q_{i_l}$ must be of the form $Q_i \wedge (x_{i_1} < x_{j_1}) \wedge (x_{i_2} \leq x_{j_2}) \wedge \ldots$ where $\pi(i_m), \pi(j_m) \in \{1, .., M\}$. Moreover, one and only one of the predicates $Q_{i_1}, Q_{i_2}, \ldots Q_{i_l}$ is true for any $\overline{x}$ that satisfies Q_i.

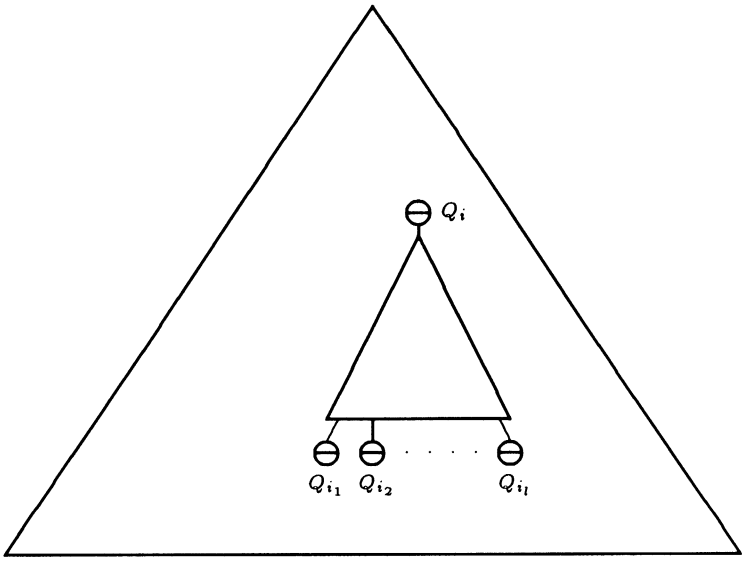

Figure 3: Comparison subtree

We now build T' from T by inductively building comparison subtrees in T' from comparison subtrees in T starting with the "uppermost" comparison subtree: The root of the new comparison subtree is the same as the root of the original comparison subtree. The internal comparison nodes are replaced with a tree that makes all the comparisons needed for a total ordering of records in internal memory. Finally, the leaves are I/O-nodes selected among the l I/O-nodes in the original subtree in the following way: If R is the

predicate "generated" on the path from the root of T' to a leaf in the new subtree, the I/O-node with the predicate Q_{i_j} such that $R \Rightarrow Q_{i_j}$ is used. The choice of I/O-node is well-defined because the predicate R implies exactly one of the Q_{i_j}'s. If any of the leaves in the original comparison subtree are also roots of comparison subtrees, i.e., they are not the leaves of T, we repeat the process for each of these subtrees. Note that any of them may appear several times in T'. It should be clear that when T' is constructed in this way, it solves P_n. Furthermore, for all $\overline{x}$, $\text{path}_{T_c}(\overline{x})$ and $\text{path}_{T'}(\overline{x})$ contain the same I/O-nodes. This means that if the height of the comparison subtrees in T' is at most h, then the number of comparison nodes on $\text{path}_{T'}(\overline{x})$ is at most $h \cdot I/O_T(\overline{x})$. But then there exists an ordinary comparison tree T_c solving P_n, such that $|\text{path}_T(\overline{x})| \leq h \cdot I/O(\overline{x})$, namely the comparison tree obtained from T' by removing the I/O-nodes.

It is obvious that our upper bound on $|\text{path}_T(\overline{x})|$ improves the smaller an h we can get. This means that we want to build a comparison tree, that after an I/O-operation determines the total order of the M records in internal memory with as small a height as possible. After an I/O-operation we know the order of the $M - B$ records that were not affected by the I/O-operation - this is an implicit invariant in the construction of T'. The problem is, therefore, limited to placing the B "new" records within this ordering. If we, furthermore, assume that we know the order of the B records, then we are left with the problem of merging two ordered lists, this can be done using at most $T_{\text{merge}}(M - B, B)$ comparisons. We cannot in general assume that the B records are ordered, but because the I/O-operations always are performed on tracks and because we know the order of the records we write to a track, the number of times we can read B records that are not ordered (and where we must use B log B comDarisons to sort them) cannot exceed $\frac{n}{B}$.

Finally, we get the desired result:

$$
\begin{array}{l}
|\text{path}_{T_c}(\overline{x})| \leq \frac{n}{B}\ B\ \text{log}B + I/O_{T'}(\overline{x}) \cdot T_{\text{merge}}(M - B, B) \\
\Downarrow \\
|\text{path}_{T_c}(\overline{x})| \leq n\ \text{log}\ B + I/O_T(\overline{x}) \cdot T_{\text{merge}}(M - B, B)
\end{array}
$$

□

Two lists of length n and m (where $n > m$) can be merged using *binary merging* [2] in $m + \lfloor \frac{n}{2^t} \rfloor - 1 + t \cdot m$ comparisons where $t = \lfloor \log \frac{n}{m} \rfloor$. This means that $T_{\text{merge}}(M - B, B) \leq B\ \log(\frac{M-B}{B}) + 3B$ which gives us the follow-

ing corollary:

Corollary 1

$$|\text{path}_{T_c}(\overline{x})| \leq n \log B + I/O_T(\overline{x}) \cdot \left(B \log(\frac{M-B}{B}) + 3B \right)$$

□

It should be clear that the corollary can be used to prove lower bounds on the number of I/O-operations needed to solve a given problem. An example is sorting, where an $n \log n - O(n)$ worst-case lower bound on the number of comparisons is known. In other words, we know that for any comparison tree (algorithm) T_c that sorts n records there is an $\overline{x}$ such that $|\text{path}_{T_c}(\overline{x})| \geq n \log n - O(n)$. From the corollary we get $n \log n - O(n) \leq n \log B + I/O_T(\overline{x}) \cdot (B \log(\frac{M-B}{B}) + 3B)$, hence the worst-case number of I/O-operations needed to sort n records is at least $\frac{n \log \frac{n}{B} - O(n)}{B \log(\frac{M-B}{B}) + 3B}$.

Note that no matter what kind of lower bound on the number of comparisons we are working with - worst-case, average or others - the theorem applies, because it relates the number of I/O's and comparisons for *each instance* of the problem.

4 Extending the Model

In this section, we discuss extensions to the operations permitted in internal memory and compare our model to the model used in [1].

The class of algorithms for which our result is valid comprises algorithms that can be simulated by our I/O-trees. This means that the only operations permitted are binary comparisons and transfers between secondary storage and internal memory. It should be obvious that a tree, using ternary comparisons and swapping of records in internal memory, can be simulated by a tree with the same I/O-height, that only uses binary comparisons and no swapping (swapping only effects the π's). Therefore, a lower bound in our model will also be a lower bound in a model where swapping and ternary comparisons are permitted. Similarly, we can permit algorithms that use integer variables, if their values are implied by the sequence of comparisons made so far,

and we can make branches according to the value of these variables. This is because such manipulations cannot save any comparisons.

The differences between our model and the model presented in [1] are, apart from ours being restricted to a comparison model, mainly three things. Firstly, Aggarwal and Vitter model parallelism with a parameter P that represents the number of blocks that can be transferred concurrently. It should be clear that we can get lower bounds in the same model by dividing lower bounds proved in our model by P. Secondly, they only assume that a transfer involves B contiguous records in secondary storage, whereas we assume that the B records constitute a track. Reading/writing across a track boundary, however, can be simulated by a constant number of "standard" I/O's. Hence, lower bounds proved in our model still apply asymptotically. Finally, their I/O's differ from ours in the sense that they permit copying of records, i.e. writing to secondary storage without deleting them from internal memory. It can be seen that the construction in the proof of our theorem still works, if we instead of one I/O-node have both an I-node and an O-node that reads from, respectively writes to, a track. Therefore, our theorem still holds when record copying is permitted.

5 Optimal Algorithms

Aggarwal and Vitter [1] show the following lower bound on the I/O-complexity of sorting:

$$\Omega\left(\frac{n \log \frac{n}{B}}{B \log \frac{M}{B}}\right)$$

They also give two algorithms based on mergesort and bucketsort that are asymptotically optimal. As mentioned earlier our result provides the same lower bound.

An almost immediate consequence of the tight lower bound on sorting is a tight lower bound on set equality, set inclusion and set disjointness, i.e., the problems of deciding whether $A = B$, $A \subseteq B$ or $A \cap B = \emptyset$ given sets A and B. It can easily be shown (see e.g. [7]) that a lower bound on the number of comparisons for each of these problems is $n \log n - O(n)$. An optimal algorithm is, therefore, to sort the two sets independently, and then solving the problem by "merging" them.

In the following, we will look at two slightly more difficult problems for which our theorem gives asymptotically tight bounds.

5.1 Duplicate Removal

We wish to remove duplicates from a file in secondary storage - that is, make a set from a multiset. Before removing the duplicates, n records reside at the beginning of the secondary storage and the internal memory is emptya The goal is to have the constructed set residing first in the secondary storage and the duplicates immediately after. Formally this corresponds to the following predicate:

$$\begin{aligned} Q_P(\overline{y}) = \exists k: \quad & (\forall i,j\ 1 \leq i,j \leq k \wedge i \neq j : y_i \neq y_j) \wedge \\ & (\forall i\ k < i \leq n : \exists j\ 1 \leq j \leq k : y_i = y_j) \end{aligned}$$

A lower bound on the number of comparisons needed to remove duplicates is $n \log n - \sum_{i=1}^{k} n_i \log n_i - O(n)$, where n_i is the multiplicity of the ith record in the set. This can be seen by observing that after the duplicate removal, the total order of the original n records is known. Any two records in the constructed set must be known not to be equal, and because we compare records using only $<$ or $\leq$ we know the relationship between them. Any other record (i.e. one of the duplicates) equals one in the set. As the total order is known, the number of comparisons made must be at least the number needed to sort the initial multiset. A lower bound on this has been shown [3] to be $n \log n - \sum_{i=1}^{k} n_i \log n_i - O(n)$.

Given this lower bound our theorem gives us the following lower bound on the I/O-complexity:

$$I/O(\text{Duplicate} - \text{Removal}_n) \in \Omega\left(\frac{n \log \frac{n}{B} - \sum_{i=1}^{k} n_i \log n_i}{B \log \frac{M}{B}}\right)$$

We match this lower bound asymptotically with an algorithm that is a variant of merge sort, where we "get rid of" duplicates as soon as we meet them. We use a block (of B records) in internal memory to accumulate duplicates, transferring them to secondary storage as soon as the block runs full.

The algorithm works like the standard merge sort algorithm. We start by making $\lceil \frac{n}{M-B} \rceil$ runs; we fill up the internal memory $\lceil \frac{n}{M-B} \rceil$ times and sort

the records, removing duplicates as described above. We then repeatedly merge c runs into one longer run until we only have one run, containing k records. $c = \lfloor \frac{M}{B} \rfloor - 2$ as we use one block for duplicates and one for the "out going" run. It is obvious that there are less than $\log_c(\lceil \frac{n}{M-B} \rceil) + 1$ phases in this merge sort, and that we in a single phase use no more than the number of records being merged times $\frac{2}{B}$ I/O-operations.

We now consider records of type x_i. In the first phase we read all the n_i records of this type. In phase j there are less than $\frac{\lceil n/(M-B) \rceil}{c^{j-1}}$ runs and we therefore have two cases:

$\frac{\lceil n/(M-B) \rceil}{c^{j-1}} \geq n_i$: There are more runs than records of the type x_i, this means that in the worst case we have not removed any duplicates, and therefore all the n_i records contribute to the I/O-complexity.

$\frac{\lceil n/(M-B) \rceil}{c^{j-1}} < n_i$ There are fewer runs than the original number of x_i's. There cannot be more than one record of the type x_i in each run and therefore the recordtype x_i contributes with no more than the number of runs to the I/O-complexity.

The solution to the equation $\frac{\lceil n/(M-B) \rceil}{c^{j-1}} = n_i$ with respect to j gives the number of phases where all n_i records might contribute to the I/O-complexity. The solution is $j = \log_{cv}(\frac{\lceil n/(M-B) \rceil}{n_i}) + 1$, and the number of times the recordtype x_i contributes to the overall I/O-complexity is no more than:

$$n_i \left(\log_c(\frac{\lceil n/(M-B) \rceil}{n_i}) + 1 \right) + \sum_{j=\log_c(\frac{\lceil n/(M-B) \rceil}{n_i})+2}^{\log_c(\lceil n/(M-B) \rceil)+1} \frac{\lceil n/(M-B) \rceil}{c^j}$$

Adding together the contributions from each of the k records we get the overall I/O-complexity:

$$\begin{aligned} I/O &\leq \frac{2}{B} \left[n + \sum_{i=1}^{k} \left(n_i \left(\log_c(\frac{\lceil n/(M-B) \rceil}{n_i}) + 1 \right) + \sum_{j=\log_c(\frac{\lceil n/(M-B) \rceil}{n_i})+2}^{\log_c(\lceil n/(M-B) \rceil)+1} \frac{\lceil n/(M-B) \rceil}{c^j} \right) \right] \\ &= \frac{2}{B} \left[n + n \log_c(\lceil n/(M-B) \rceil - \sum_{i=1}^{k} n_i \log_c n_i + n + \right. \\ &\quad \left. \sum_{i=1}^{k} \lceil n/(M-B) \rceil \cdot \left(\sum_{j=0}^{\log_c(\lceil n/(M-B) \rceil)+1} (c^{-1})^j - \sum_{j=0}^{\log_c(\frac{\lceil n/(M-B) \rceil}{n_i})+1} (c^{-1})^j \right) \right] \end{aligned}$$

$$
\begin{aligned}
&= \frac{2}{B}\left[n + n\,\log_c(\lceil n/(M-B)\rceil) - \sum_{i=1}^{k} n_i\,\log_c\,n_i + n + \frac{n-k}{c^2-c}\right] \\
&= 2\frac{n\,\log(\lceil n/(M-B)\rceil) - \sum_{i=1}^{k} n_i\,\log\,n_i}{B\,\log(\lceil \frac{M}{B}\rceil - 2)} + \frac{4n}{B} + \frac{2(n-k)}{B(c^2-c)} \\
&\in O\left(\frac{n\,\log\,\frac{n}{B} - \sum_{i=1}^{k} n_i\,\log\,n_i}{B\,\log\frac{M}{B}}\right)
\end{aligned}
$$

5.2 Determining the Mode

We wish to determine the mode of a multiset, i.e. the most frequently occurring record. In a start configuration, the n records reside at the beginning of the secondary storage. The goal is to have an instance of the most frequently occurring record residing first in secondary storage and all other records immediately after. Formally this corresponds to the following predicate:

$$Q_P(\overline{y}) = \forall j\ 1 \le j \le n : |\{i \mid y_i = y_1, 1 \le i \le n\}| \ge |\{i \mid y_i = y_j, 1 \le i \le n\}|$$

Munro and Raman [3] showed that $n\,\log\,\frac{n}{a} - O(n)$ is a lower bound on the number of ternary comparisons needed to determine the mode, where a denotes the frequency of the mode. This must also be a lower bound on the number of binary comparisons, thus, our theorem gives the following lower bound on the number of I/O-operations:

$$I/O(\text{mode}_n) \in \Omega\left(\frac{n\,\log\,\frac{n}{aB}}{B\,\log\,\frac{M}{B}}\right)$$

The algorithm that matches this bound is inspired by the distribution sort algorithm presented by Munro and Spira [4]. First, we divide the multiset into c disjoint segments of roughly equal size (a segment is a sub-multiset which contains all elements within a given range). We then look at each segment and determine which records (if any) have multiplicity greater than the segment size divided by a constant l (we call this an l-majorant). If no segments contained an l-majorant, the process is repeated on each of the segments. If, on the other hand, there were any l-majorants, we check

whether the one among these with the greatest multiplicity has multiplicity greater than the size of *the largest segment* divided by l. If it does, we have found the mode. If not, we continue the process on each of the segments as described above.

We now argue that both the division into segments and the determination of l-majorants can be done in a constant number of sequential runs through each segment.

To determine l-majorants we use an algorithm due to Misra and Gries [5]. First, $l-1$ candidates are found in a sequential run through the segment in the following way: For each record it is checked whether it is among the present $l-1$ candidates (initially each of the $l-1$ candidates are just "empty"). If it is, this candidates multiplicity is incremented by $l-1$. If not, all the candidates multiplicities are decremented by 1, unless any of the candidates had multiplicity 0 (or was "empty"), in which case the record becomes a candidate with multiplicity $l-1$ instead of one with multiplicity 0. When the run is completed, if there were any l-majorants, they will be among the candidates with positive multiplicity. This is checked in another sequential run, where the actual multiplicities of the candidates are determined. Note that l must be less than $M-B$ because the l candidates have to be in internal memory.

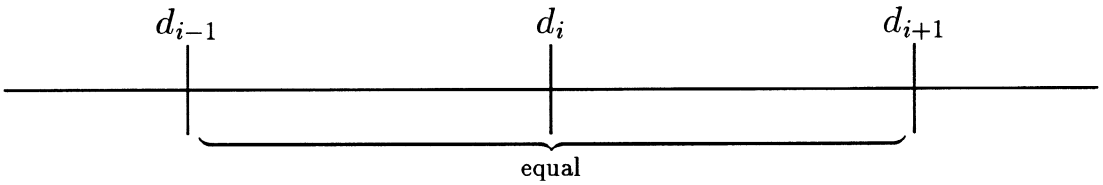

Figure 4: "The worst case": $d_{i-1} \neq d_i \neq d_{i+1}$ and $[d_{i-1}, d_i[$ contains one element whereas $[d_i, d_{i+1}[$ is double the intended size.

The division of a segment into c disjoint segments of roughly equal size is done by first finding c pivot elements, and then distributing the records in the original segment into the segments defined by the pivot elements in a sequential run. We use one block in internal memory for the ingoing records, and c blocks, one for each segment, for the outgoing records (this means $c \leq \lfloor \frac{M}{B} \rfloor - 1$). Aggarwal and Vitter [1] describe an algorithm to find $\sqrt{\frac{M}{B}}$ pivot elements in a *set* in $O(\frac{n}{B})$ I/O's. The algorithm satisfies the following:

If $d_1, d_2, \ldots, d_{\sqrt{\frac{M}{B}}}$ denote the pivot elements and K_i denotes the elements in $[d_{i-1}, d_i[$ then $\frac{n}{2\sqrt{\frac{M}{B}}} \leq |K_i| \leq \frac{3n}{2\sqrt{\frac{M}{B}}}$ $(\star)$. We now wish to argue that a slightly modified version of the algorithm can be used to find pivot elements in a *multiset* so that either $|K_i| \leq \frac{3n}{\sqrt{\frac{M}{B}}}$ or else all the records in K_i are equal.

We start by using the algorithm by Aggarwal and Vitter. This algorithm depends almost exclusively on the *k-selection* algorithm that finds the kth smallest element in a multiset in linear time [6]. This means that if we implicitly assume an order of equal records, namely the order in which we meet them, the resulting pivot elements define segments that satisfy $(\star)$. Some of the pivot elements might be equal, and we therefore use some slightly different elements. If $d_{i-1} \neq d_i = d_{i+1} = \ldots = d_{i+k} \neq d_{i+k+1}$, we use the elements $d_{i-1}, d_i, succ(d_{i+k})$ and d_{i+k+1}, where $succ(d)$ is the successor to d in the record order. The segments now either consist of all equal records, or else they are no more than double the size of the segments we get, assuming that all records are distinct. This can be seen by looking at the "worst case" which is when $d_{i-1} \neq d_i \neq d_{i+1}$ and all records in $]d_{i-1}, d_{i+1}[$ are equal (see figure 4).

We have argued that the number of I/O-operations at each level is proportional to n/B, and the analysis therefore reduces to bounding the number of levels. An upper bound on the size of the largest segment on level j must be $\frac{n}{(\frac{1}{3}\sqrt{M/B})^j}$. It follows that the algorithm can run no longer than to a level j where $\frac{n}{(\frac{1}{3}\sqrt{M/B})^j}/l \leq a$. Solving this inequality with respect to j, we find that no matter what value of l we choose in the range $\{B, \ldots, M - B\}$, we get a bound on the number levels of $O(\frac{\log \frac{n}{aB}}{\log \frac{M}{B}})$. This gives us the matching upper bound of $O(\frac{n \log \frac{n}{aB}}{B \log \frac{M}{B}})$.

Acknowledgments

The authors thank Gudmund S. Frandsen, Peter Bro Miltersen and Erik Meineche Schmidt for valuable help and inspiration. Special thanks go to Peter Bro Miltersen and Erik Meineche Schmidt for carefully reading drafts of this paper and providing constructive criticism.

References

[1] Aggarwal, A., Vitter, J.S.: The I/O Complexity of Sorting and Related Problems. Proc 14th ICALP (1987), Lecture Notes in Computer Science 267, Springer Verlag, 467-478, and: The Input/Output Complexity of Sorting and Related Problems. Communications of the ACM, Vol 31 (9) (1988) 1116-1127.

[2] Knuth, D.E.: The Art of Computer Programming, Vol 3: Sorting and Searching, Addison-Wesley (1973) (p. 205-206).

[3] Munro, J.I., Raman, V.: Sorting Multisets and Vectors In-Place. Proceedings of WADS '91, Lecture Notes in Computer Science 519, Springer Verlag (1991), 473-479.

[4] Munro, I., Spira, P.M.: Sorting and Searching in Multisets. SIAM Journal of Computing, 5 (1) (1976) 1-8.

[5] Misra, J., Gries, D.: Finding Repeated Elements. Science of Computer Programming 2 (1982), 143-152, North-Holland Publishing.

[6] Blum, M., Floyd, R.W, Pratt, V.R, Rivest, R.L, Tarjan, R.E.: Time bounds for selection. Journal of Computer and System Sciences, 7(4) (1972), 448-461.

[7] Ben-Or, M.: Lower bounds for algebraic computation trees. Proc. 15th Ann. ACM Symp. on Theory of Computation, MA (1983) 80-86.