Lecture Notes: Comparison-Based Lower Bounds

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

We have proved in a previous lecture that the permutation problem on n elements requires $\Omega(\min\{n, \frac{n}{B}\log_{M/B}\frac{n}{B}\})$ I/Os to solve in the indivisibility model. This immediately implies the same lower bound for sorting—if we can sort n elements in x I/Os, then we can also permute them in $O(x)$ I/Os. This, in turn, means that no "indivisible" EM algorithm can *always* sort in $o(\frac{n}{B}\log_{M/B}\frac{n}{B})$ I/Os, because this will violate the aforementioned lower bound when $\log_{M/B}\frac{n}{B} = O(B)$.

How about $\log_{M/B}\frac{n}{B} = \omega(B)$? Is it possible to sort in $o(\frac{n}{B}\log_{M/B}\frac{n}{B})$ in this case? In the indivisiblity model, the answer is yes—there is a trivial, but unrealistic, algorithm to perform sorting $O(n)$ I/Os (think: how?). However, in the EM model, no algorithm is known to be able to achieve the purpose. This raises an intriguing question—is there an ingenious $O(n)$-cost EM algorithm yet to be discovered, or is the indivisibility model simply too powerful in the scenario $\log_{M/B}\frac{n}{B} = \omega(B)$? This is still an open question to this day.

In this lecture, we will hit a middle ground between the EM and indivisibility models. We will define a new model called the *I/O comparison model*, which is more restrictive than the indivisibility model (in terms of what an algorithm can do). This new model still captures a broad class of EM algorithms, known as the *comparison class*. We will see that any algorithm in the I/O comparison model must perform $\Omega(\frac{n}{B}\log_{M/B}\frac{n}{B})$ I/Os to sort, for *all* values of n, B, and M.

1 Review: Comparison-Based Algorithms in Internal Memory

Let us re-visit the *comparison model* in internal memory for lower bound analysis. Let S be a set of n elements, denoted as $e_1, e_2, ..., e_n$, respectively. Element e_i ($1 \le i \le n$) is said to have *id* i. An algorithm under this model—referred to as a *comparison-based algorithm*—can be described by a binary *decision tree* T, defined as follows. Each internal node u of T has two child nodes, and is associated with two element ids i, j. When the algorithm is at u, it always performs a comparison between e_i and e_j. If $e_i < e_j$, the algorithm moves to the left child of u; otherwise ($e_i > e_j$), the algorithm moves to the right child of u. A leaf node z of T is associated with an answer, which is the final output of the algorithm when it reaches z. The algorithm must always start from the root of T.

It is important to note that T is given *before* seeing S. In other words, regardless of the contents of $e_1, e_2, ..., e_n$, the algorithm must always behave according to the *same* T. The *cost* of the algorithm is the height of T.

It is usually quite straightforward to lower bound the cost of comparison-based algorithms—all we need to do is to see how many different answers an algorithm must be able to produce. For example, for sorting, an algorithm must produce $n!$ different answers. This means that T must have at least $n!$ leaves. Hence, the height of T must be at least $\log_2 n! = \Omega(n \log n)$.

2 The I/O Comparison Model

We are now ready to define the *I/O comparison model*. Let S be a set of n distinct elements $e_1, ..., e_n$. The memory is a set of at most M elements in S, while the disk is a sequence of *blocks*, each of which is a set of B elements. The values of M and B satisfy $M \geq 2B$. S is given in a disk-resident array storing the sequence $e_1, e_2, ..., e_n$.

An algorithm is modeled as an *I/O decision tree* T_{IO} defined as follows. Each internal node u of T_{IO} is associated with a set $I(u)$ of at most M element ids, corresponding to the at most M elements in memory. Furthermore, u belongs to one of the following three types:

- *Read node:* In this case, u is also associated with an integer $addr(u) \geq 1$. When the algorithm is at u, it loads the $addr(u)$-th disk block into memory, perhaps overwriting some existing elements in memory. Node u has only one child u' such that $I(u')$ is the set of ids of the elements in memory after the read.

- *Write node:* In this case, u is also associated with an integer $addr(u) \geq 1$, and an operation tag $op(u)$ whose value can be "`c`" or "`r`". If $op(u) = \texttt{c}$, the algorithm (when at u) selects B elements from memory, and writes them into a new block, and make the block the $addr(u)$-th one in the disk. If $op(u) = \texttt{r}$, the algorithm selects B elements from memory, and writes them into the $addr(u)$-th disk block (the original contents of the block are erased). In any case, u has only one child u' with $I(u) = I(u')$.

- *Comparison node:* It sorts the at most M elements whose ids are in $I(u)$, and descends into a *different* child for each different ordered permutation of those elements. For each child node u', $I(u') = I(u)$.

If u is the root, then u must be a read node with $I(u) = \emptyset$. We will refer to read and write nodes collectively as *I/O nodes*. Finally, each leaf node z is associated with an answer, which is the final output of the algorithm when it reaches z.

T_{IO} is given before seeing S, namely, the algorithm must behave according to T_{IO} regardless of the contents of $e_1, ..., e_n$. In particular, note that the id-set $I(u)$ of each node u is a part of T_{IO}, and hence, does *not* change with S. In other words, whenever the algorithm reaches u, the memory-resident elements always have the same ids. Similarly, the fields $op(u)$ and $addr(u)$, if applicable, do not change with S, either. The *cost* of the algorithm is the maximum number of I/O nodes along any root-to-leaf path of T_{IO}. Note that comparison nodes are not counted.

It is not hard to observe that the I/O comparison model is subsumed by the indivisibility model. Specifically, given an algorithm in the former model that performs x I/Os, we can easily obtain an algorithm in the latter model that performs x I/Os. The opposite, however, is not true. As noted earlier, there is a trivial $O(n)$-cost sorting algorithm in the indivisibility model. In the next section, we will prove the absence of such an algorithm in the I/O comparison model.

Neither the I/O comparison model nor the EM model subsumes the other. A lower bound in the I/O comparison model only applies to a set of EM algorithms that can be implemented in the I/O comparison model—the set is known as the *comparison class*. It is worth mentioning that both external sort and distribution sort belong to this class.

3 Sorting Lower Bound in the I/O Comparison Model

Next, we prove that $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ is a sorting lower bound in the I/O comparison model. Our argument is mainly due to Aggarwal and Vitter [1] with some ideas from Erickson [2]. We will need the following basic property of I/O decision trees:

Lemma 1. *Let u be a node in an I/O decision tree T_{IO}, and $I^\star$ any subset of $I(u)$. Suppose that there is a comparison node v such that (i) v is a proper ancestor of u, and (ii) $I^\star \subseteq I(v)$. Then, the relative ordering of the elements with ids in $I^\star$ is fixed at u.*

Proof. By definition of T_{IO}, when the algorithm descends from v (to any of its child nodes), it determines the relative ordering of the elements with ids in $I(v)$. $\square$

3.1 Regular I/O Decision Trees

Let us first prove the lower bound on *regular* I/O decision trees T_{IO}. Specifically, we say that T_{IO} is *regular* if all the following hold:

- every node is *useful*, namely, it can be reached by at least one input S;
- every read node is the parent of a comparison node;
- the parent of every comparison node is a read node;
- on any root-to-leaf path of T_{IO}, the first $2n/B$ nodes must implement an *initialization phase* as follows. Denote those nodes as $u_1, v_1, u_2, v_2, ..., u_{n/B}, v_{n/B}$, in the order the appear on the path. Each u_i ($1 \le i \le n/B$) is a read node, and each v_i is a comparison node. Furthermore, node u_i reads the i-th input block of S, and makes sure that, after the read, the memory contains only the B elements of that block (i.e., these elements will then be overwritten after the next input block is read).

Since there must be at least $n!$ leaves and each comparison node obviously has a fanout at most $M!$, at least one root-to-leaf path of T_{IO} has $\Omega(\log_{M!} n!) = \Omega(\frac{n \log n}{M \log M})$ comparison nodes. By the regularity of T_{IO}, there are $\Omega(\frac{n}{M} \log_M n)$ read nodes on that path. This is already a lower bound on the cost of T_{IO}, but it is much looser than what we aim for. The cause of looseness is that we have *severely* over-estimated how many child nodes a comparison node can have.

This is particularly obvious for each comparison node v during the initialization phase. Note that $I(v)$ has a size of B. Therefore, v can have only $B!$ child nodes, much less than $M!$.

Let us now consider any comparison node v *after* the initialization phase. Denote by u the parent of v (i.e., u is a read node). Define I_1, I_2 as follows:

$$\begin{aligned} I_2 &= \text{the set of ids of the } B \text{ elements read by } u \\ I_1 &= I(v) \setminus I_2 \end{aligned}$$

We now prove a crucial fact:

Lemma 2. *The relative ordering of the elements in I_1 is fixed at node v. The same is true with respect to the elements in I_2.*

Proof. We will first prove the claim about I_1. Simply consider the lowest comparison node $\hat{v}$ that is a proper ancestor of v. Note that $\hat{v}$ always exists because v is after the initialization phase. It is clear that $I(u) = I(\hat{v})$ because the path from $\hat{v}$ to u contains nothing but write nodes. Note also that $I_1 \subseteq I(u)$ because every id in $I(v)$ must belong to either I_2 or $I(u)$. This means that $I_1 \subseteq I(\hat{v})$. Hence, by Lemma 1, the claim is true about I_1.

Let us now look at I_2. As u occurs after the initialization phase, the block it reads is either an input block of S, or a block written by the algorithm itself. It thus follows that the elements with ids in I_2 have co-existed in memory before. Let $\hat{u}$ be the read node that made this happen for the first time, and $\hat{v}$ the child of $\hat{u}$. It thus follows that $I_2 \subseteq I(\hat{v})$ and $\hat{v}$ is a proper ancestor of v. Hence, by Lemma 1, the claim is also true about I_2. $\square$

With the above said, we can take the view that each ordered permutation of the elements with ids in $I(u)$ is created in three steps:

1. Line up $|I_1| + |I_2|$ empty slots.
2. Choose $|I_1|$ empty slots, and place the elements with ids in I_1 at those slots, according to their already-determined ordering.
3. Place the elements with ids in I_2 at the remaining slots, according to their already-determined ordering.

The maximum number of permutations that can be created equals $\binom{|I_1|+|I_2|}{|I_2|}$, which is at most $\binom{M}{B}$ given that $|I_1| + |I_2| \leq M$, $|I_2| = B$, and $M \geq 2B$. In other words, the fanout of v is at most $\binom{M}{B}$.

Consider the moment when the algorithm has just finished the initialization phase on S. Let $u^\star$ be the node that the algorithm is standing at currently. The subtree of $u^\star$ must have at least $n!/(B!)^{n/B}$ leaves (think: why?). Therefore, at least one $u^\star$-to-leaf path has at least the following number of comparison nodes:

$$\begin{aligned}
\log_{\binom{M}{B}} \frac{n!}{(B!)^{n/B}} &= \frac{\log n! - \log(B!)^{n/B}}{\log \binom{M}{B}} \\
&= \Omega\left(\frac{n}{B} \log_{M/B} \frac{n}{B}\right).
\end{aligned}$$

By regularity, there must be the same number of read nodes on that path.

3.2 Reduction to Regular I/O Decision Trees

It remains to discuss I/O decision trees T_{IO} that are not regular. Suppose that T_{IO} has cost x. We will convert it into a regular I/O decision tree T'_{IO} with cost $x + n/B$. Our result in the previous section suggests that $x + n/B = \Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$. It thus follows that $x = \Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$.

Let A be the algorithm described by T_{IO}. Our conservation is based on several principles. First, we impose a compulsory initialization phase before starting to execute A. Second, whenever A executes a read node, we always force the execution of a comparison node—doing so allows us to perform the largest number of comparisons possible. Third, we follow every write node of A faithfully. Next, we explain the details.

We start by building up the first $2n/B$ levels of T'_{IO} that correspond to the initialization phase. Each node v at the $(2n/B)$-th level (the root is at level 1) is a comparison node with $B!$ child nodes. We copy the entire T_{IO} to be the subtree rooted at each child node of v. Note that T_{IO} is copied $(B!)^{n/B}$ times this way. Our current T'_{IO} is now an I/O decision tree correctly solving the sorting problem (think: why? Hint: take an input S, and see which leaf the algorithm will fall into).

T'_{IO} may not be regular at this point. We can make it so with a depth-first traversal of the subtree of each node at the $(1 + 2n/B)$-th level (this subtree is a copy of T_{IO}). Let u the node we are currently at:

- Case 1: u is a read node. Let T' be the subtree of u. We remove T', and create a child comparison node v for u. Then, for each possible ordered permutation of the elements with ids in $I(v)$, create a child for v, and copy the entire T' to be the subtree rooted at this child (T' is copied as many times as the number of children of v). Continue the depth-first traversal into the left-most subtree of v.
- Case 2: u is a write node. No change; simply continue the traversal.
- Case 3: u is a comparison node. Only one child—say u'—of u is useful now (think: why). Remove u from T'_{IO}, and make u' the only child of the parent of u.

The T'_{IO} thus constructed fulfills our purposes (think: why? Hint: prove the correctness inductively).

4 Remarks

We have defined the comparison model in internal memory and external memory by assuming that the input set S has n distinct elements. This is purely for the convenience of studying sorting. Comparison-based algorithms are also popular for solving other problems that involve identical elements. One example is the *element distinctness problem*, where the input is a multi-set S of n elements, and the goal is to decide whether two elements in S are identical (namely, if S is a set or really a multi-set). Note that the decision tree defined in Section 1 is *not* appropriate for solving the element distinctness problem, because we have not defined what to do when a node finds out $e_i = e_j$, where e_i, e_j are the two elements compared at the node. However, this issue can be easily fixed by defining a slightly different decision tree (and hence, a slightly different comparison model), where each node has three branches, instead of two. One can show that any such decision tree must incur $\Omega(n \log n)$ comparisons to solve the element distinctness problem. Similar extensions can also be made to I/O decision trees. Such an extended I/O decision tree needs to perform $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os to solve the element distinctness problem.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.

[2] J. Erickson. Lower bounds for external algebraic decision trees. In *SODA*, pages 755–761, 2005.