Lecture 21 — Sorting Lower Bounds and How to Beat Them

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Kanat Tangwongsan — April 3, 2012

1 Lower Bounds

After spending time formulating a concrete problem, we might wonder how hard the problem actually is. In this course thus far, our focus has been on obtaining efficient algorithms for certain problems. For a problem P, we try to design efficient algorithms to solve it. The existence of an algorithm gives an upper bound on the complexity of the problem P. In particular, an algorithm A with work (either expected or worst-case) $O(f(n))$ is a constructive proof that P can be solved provided $O(f(n))$ work. This is essentially the upper bound part of the question.

In this lecture, we'll turn the tables, showing that certain problems cannot be solved more efficiently than a given bound. This is the lower bound part of the question. In general, this is a harder task: To establish a lower bound, we have to argue that *no algorithm, however smart, can possibly do better than what we claim*; it is no longer sufficient to exhibit an algorithm A and analyze its performance.

1.1 Sorting and Merging Lower Bounds

Before we look at lower bounds for sorting and merging, let us review the (upper) bounds we have for various sorting algorithms we've covered:

Algorithm	Work	Span
Quick Sort	$O(n\log n)$	$O(\log^2 n)$
Merge Sort	$O(n\log n)$	$O(\log^2 n)$
Heap Sort	$O(n\log n)$	$O(n\log n)$
Balanced BST Sort	$O(n\log n)$	$O(\log^2 n)$

Notice that in this table, all algorithms have $O(n\log n)$ work—and except for heap sort, every algorithm is very parallel ($\log^2 n$ span). Can we sort in less than $O(n\log n)$ work? Probably. But we'll show that *any* deterministic *comparison-based* sorting algorithm must use $\Omega(n\log n)$ comparisons to sort n entries in the worst case. In the comparison-based model, we have no domain knowledge about the entries and the only operation we have to determine the relative order of a pair of entries x and y is a comparision operation, which returns whether $x < y$. More precisely, we'll prove the following theorem:

Theorem 1.1. *For a sequence $\langle x_1, \ldots, x_n\rangle$ of n distinct entries, finding the permutation π on $[n]$ such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$ requires, in the worst case, at least $\frac{n}{2}\log(\frac{n}{2})$ queries to the $<$ operator.*

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

Since each comparison takes at least constant work, this implies an $\Omega(n \log n)$ lower bound on the work required to sort a sequence of length n in the comparison model.

What about merging? Can we merge sorted sequences faster than resorting them? As seen in previous lectures, we can actually merge two sorted sequences in $O(m \log(1 + n/m))$ work, where m is the length of the shorter of the two sequences, and n the length of the longer one. We'll show, however, that in the comparision-based model, we cannot hope to do better:

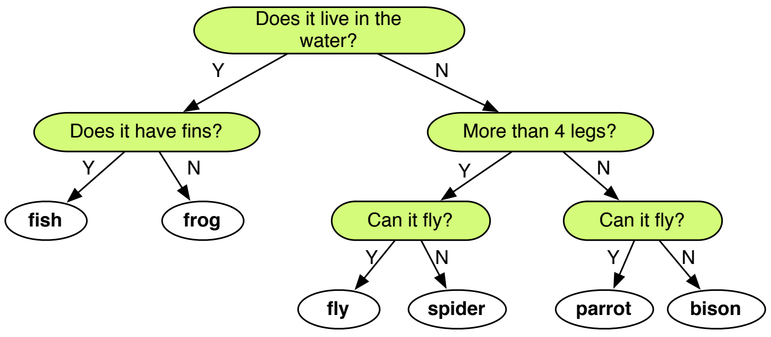
Theorem 1.2. *Merging two sorted sequences of lengths m and n ($m \le n$) requires at least*

$$m \log_2(1 + \tfrac{n}{m})$$

comparison queries in the worst case.

1.2 Decision Trees or The 20 Questions Game

Let's play game. Suppose I think of an animal but you know for fact it's one of the following: a fish, a frog, a fly, a spider, a parrot, or a bison. You want to find out what animal that is by answering the fewest number of Yes/No questions (you're only allowed to ask Yes/No questions). What strategy would you use? Perhaps, you might try the following reasoning process:



Interestingly, this strategy is optimal: There is no way you could have asked any 2 Yes/No questions to tell apart the 6 possible answers. If we can ask only 2 questions, any strategy that is determistic and computes the output using only the answers to these Yes/No questions can distinguish between only $2^2 = 4$ possibilities. Thus, using 3 questions is the best one can do.

Determining the minimum number of questions necessary in the worst case in at the crux of many lower-bound arguments. For starters, we describe a way to represent a deterministic strategy for playing such a game in the definition below.

Definition 1.3 (Binary Decision Trees)**.** A *decision tree* is a tree in which

- each leaf node is an answer (i.e. what the algorithm outputs);
- each internal node represents a query—some question about the input instance—and has k children, corresponding to one of the k possible responses $\{0, \ldots, k-1\}$;
- and the answer is computed as follows: we start from the root and follow a path down to a leaf where at each node, we choose which child to follow based on the query response.

The cruical observation is the following: If we're allowed to make at most q queries (i.e., ask at most q Yes/No questions), the number of possible answers we can distinguish is the number of leaves in a binary tree with depth at most q; this is at most 2^q. Taking logs on both sides, we have

$$\boxed{\text{If there are } N \text{ possible outcomes, the number of questions needed is at least } \log_2 N.}$$

That is, there is *some* outcome, that requires answering at least $\log_2 N$ questions to determine that outcome.

1.3 Warm-up: Guess a Number

As a warm-up question, if I pick a number a between 1 and 2^{20}, how many Yes/No questions you need to ask before you can zero in on a? By the calculation above, since there are $N = 2^{20}$ possible outcomes, you will need *at least*

$$\log_2 N = 20$$

questions in the worst case.

Another way to look at the problem is to suppose I am devious and I don't actually pick a number in advance. Each time you ask a question of the form "is the number greater than x", in effect you are splitting the set of possible numbers into two groups. I always answer so the set of remaining possible numbers has the greater cardinality. That is, each question you ask eliminates at most half of the numbers. Since there are $N = 2^{20}$ possible values, I can force you ask $\log_2 N = 20$ questions before I must concede and pick the last remaining number as my a. This variation of the games shows that no matter what strategy you use to ask questions, there is always *some* a that would cause you to ask a lot of questions.

1.4 A Sorting Lower Bound

Let's turn back to the classical sorting problem. We will prove Theorem 1.1. This theorem follows almost immediately from our observation about k-ary decision trees. There are $n!$ possible permutations, and to narrow it down to one permutation which orders this sequence correctly, we'll need $\log(n!)$ queries, so the number of comparison queries is at least

$$\begin{aligned}\log(n!) &= \log n + \log(n-1) + \cdots + \log(n/2) + \cdots + \log 1\\ &\geq \log n + \log(n-1) + \cdots + \log(n/2)\\ &\geq \tfrac{n}{2} \cdot \log(n/2).\end{aligned}$$

We can further improve the constants by applying Stirling's formula instead of this crude approximation. Remember that *Stirling's formula* gives the following approximation:

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta(n^{-1})\right) > \left(\frac{n}{e}\right)^n,$$

so $\log_2(n!) > n \log_2(n/e)$.

1.5 A Merging Lower Bound

Closely related to the sorting problem is the merging problem: Given two sorted sequences A and B, the merging problem is to combine these sequences into a sorted one. To apply the argument we used for sorting, we'll need to count how many possible outcomes the comparsion operation can produce.

Suppose $n = |A|$, $m = |B|$, and $m \leq n$. We'll also assume that these sequences are made up of unique elements. Now observe that we have not made any comparision between elements of A and B. This means any interleaving sequence A's and B's elements is possible. Therefore, the number of possible merged outcomes is the number of ways to choose n positions out from $n + m$ positions to put A's elements; this is simply $\binom{n+m}{n}$. Hence, we'll need, in the worst case, at least $\log_2 \binom{n+m}{n}$ comparison queries to merge these sequences.

The following lemma gives a simple lower bound for $\binom{n}{r}$, so that we can simplify $\log_2 \binom{n+m}{n}$ to an expression that we recognize.

Lemma 1.4 (Binomial Lower Bound)**.**

$$\binom{n}{r} \geq \left(\frac{n}{r}\right)^r.$$

Proof. First, we recall that

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)\ldots(n-r+1)}{r(r-1)(r-2)\ldots 1} = \prod_{i=0}^{r-1} \frac{n-i}{r-i}.$$

We'll argue that for $0 \leq i < \min(r, n)$, $\frac{n-i}{r-i} \geq \frac{n}{r}$. Notice that

$$\frac{n-i}{r-i} \geq \frac{n}{r} \iff r(n-i) \geq n(r-i) \iff rn - ri \geq nr - ni \iff (n-r)i \geq 0.$$

Therefore, we have $\binom{n}{r} \geq \prod_{i=0}^{r-1} \frac{n-i}{r-i} \geq (n/r)^r$. □

With this lemma, we conclude that the number of comparison queries needed to merge sequences of lengths m and n ($m \leq n$) is at least

$$\log_2 \binom{n+m}{m} \geq m \log_2 \left(1 + \tfrac{n}{m}\right),$$

proving Theorem 1.2

2 Radix Sort

The lower bound we just proved only tells us that we can't hope to do better than $\Omega(n \log n)$ if we want a comparison-sort algorithm. It doesn't rule out the possibility of obtaining a faster algorithm when we have extra knowledge about the elements, for example, when sorting integer keys. An example of algorithms that take advantage of such information is radix sort, which we're going to discuss next.

Say we are given a sequence of n integers ranging between 1 and U, where U is some fixed upper-bound, and we want to sort this sequence. Comparison-based sorting algorithms such quick sort and merge sort can sort this sequence in $O(n \log n)$ work and $O(\log n)$ span. Let's first focus on sequential algorithms: An alternative to these algorithms is the radix sort algorithm, which dates back to as far as 1887. There are actually many variants of radix sort; we are going to discuss a version that processes the digits from the least significant to the most significant. In this version, the input sequence is scanned from right to left: for each position, it groups together keys based on that digit but otherwise retains the original order. That is, we have the following algorithm:

```
Input: S - a sequence of d-digit numbers
For i = d-1 to 0:
  Sort S by position i (in case of ties, keep original ordering)
```

Let's look at an example:

```
251        210          1         1
122        251        210        42
210        451        122       122
451  ==>     1  ==>    42  =>   210
384        122        251       251
499         42        451       384
  1        384        859       451
 42        499        384       499
859        859        499       859
```

How do we sort each digit? It is crucial that we maintain the original ordering in case of ties; consider the following example: let's sort 57, 52, 1.

```
57         1        1
52  ==>   52  ==>  57
 1        57       52
```

We would like to be able to sort a sequence of n numbers by its i-th digit in $O(n)$ time. This is in fact easy to do, for example, by keeping queues for each possible digit value. That is, to sort digit i, we go through the sequence in order and for each element e, we enqueue it to the queue corresponding to the value e_i. Finally, we concatenate the queues together.

```
for j = 0 to n - 1:
  Q[S[j][i]] = enqueue(Q[S[j][i]], j)
S’ = []
for k = 0 to 9:
  S’ = append(S, Q[k])
```

This process costs us $O(n)$ to sort each position, so since there are d positions to sort, the radix sort algorithm is $O(nd)$—and the span is also $O(nd)$.

So this is sequential!

3 Parallel Integer Sort

Theorem 3.1. *There is an algorithm* `parallelIntSort` *that on input a sequence* $S = \langle s_1, \ldots, s_n \rangle$ *where* $s_i \in [n]$*, sorts the sequence* S *in* $O(n)$ *work and* $O(\log^2 n)$ *span.*