Lecture 20: Models of Computation, Selection, and Sorting

Objectives of this lecture

- Understand why the choice of computational model matters for both upper and lower bounds.
- See a deterministic $O(n)$-time algorithm for selecting the k-th smallest element.
- Prove that comparison-based sorting requires $\Omega(n \log n)$ comparisons.
- See how counting sort and radix sort can beat the comparison lower bound in stronger models.

1 Philosophy

A running-time bound is only meaningful after we specify what the algorithm is allowed to do in one step. There are two models to keep in mind today.

Comparison model. The input consists of keys, and the algorithm is only allowed to learn about them by asking comparison questions such as: is $x_i < x_j$, where x_i and x_j are inputs. This model is restrictive, but it is great for proving lower bounds because of its simplicity.

Word-RAM model. The machine stores $O(\log n)$-bit words, and standard arithmetic and indexing operations on a word take $O(1)$ time. So if the input numbers are polynomially bounded in n, they fit in one word and can be manipulated directly.

The point of today's lecture is that lower bounds in the comparison model are *not* universal lower bounds for all models of computation. In other words, you have to be very careful about specifying the model when trying to understand the complexity of problems. Specifically, we will show:

- selection is easier than sorting in the comparison model;
- sorting needs $\Omega(n \log n)$ comparisons in the comparison model;
- but in word-RAM, integer sorting can be asymptotically faster by using the bit representation of the numbers.

2 Deterministic Linear-Time Selection

Problem: Selection

Given $x_1, \ldots, x_n$ and an integer $k \in \{1, \ldots, n\}$, find the k-th smallest element using only $O(n)$ comparisons.

A simple idea is to sort first and then return the k-th element, but that costs $O(n \log n)$ comparisons. Interestingly, this *selection* problem can be done deterministically in linear time.

The median-of-medians algorithm

For simplicity, first assume that all keys are distinct and that n is a multiple of 5. These assumptions are only for exposition; the general case changes the constants but not the asymptotic running time.

1. Partition the n elements into $g = n/5$ groups of size 5.
2. Find the median of each group. Call these medians

$$m_1, m_2, \ldots, m_g.$$

   Since each group has constant size, this step takes $O(n)$ total time.
3. Recursively find the median of the list $m_1, \ldots, m_g$. Call it q.
4. Use q as a pivot and partition the input into

$$L = \{x_i : x_i < q\}, \qquad E = \{x_i : x_i = q\}, \qquad G = \{x_i : x_i > q\}.$$

5. If $k \le |L|$, recurse on L. If $|L| < k \le |L| + |E|$, return q. Otherwise recurse on G, with the obvious adjusted rank.

The whole point is that the pivot q is guaranteed to be good enough that we throw away a constant fraction of the elements each time.

Why is the pivot good?

We will show that both sides of the partition are not too large.

Among the g medians $m_1, \ldots, m_g$, at least $\lfloor g/2 \rfloor$ are at most q, and at least $\lfloor g/2 \rfloor$ are at least q. Ignore the one group whose median is exactly q. Then at least

$$\lfloor g/2 \rfloor - 1$$

groups have median strictly less than q.

Now fix one such group. If its median is less than q, then in that group there are at least 3 elements less than q: the median itself and the two elements below it. Therefore the total

number of elements less than q is at least

$$3\big(\lfloor g/2\rfloor - 1\big) = \frac{3n}{10} - O(1).$$

By the same argument, the number of elements greater than q is also at least

$$\frac{3n}{10} - O(1).$$

So after partitioning around q, the recursive call is on a set of size at most

$$n - \left(\frac{3n}{10} - O(1)\right) = \frac{7n}{10} + O(1).$$

Running time

The recursive call to find the pivot uses only the $g = n/5$ medians, and the recursive call after partitioning uses at most $7n/10 + O(1)$ elements. Everything else is linear work. Thus

$$T(n) \le T(\lceil n/5\rceil) + T(7n/10 + O(1)) + O(n).$$

Since $\frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$, the *master theorem* gives that $T(n) = O(n)$.

Why groups of five? This is the smallest group size that makes the analysis work cleanly. If we grouped into threes, we would get a recurrence of the form

$$T(n) \le T(n/3) + T(2n/3) + O(n),$$

and the subproblem fractions add up to 1, which leads to $O(n \log n)$ rather than $O(n)$. Grouping by five is the first choice that forces the sum of the recursive fractions below 1. You could also use bigger groups (like size 7, etc.).

3 Sorting Lower Bounds in the Comparison Model

Problem: Sorting lower bound

Any deterministic comparison-based algorithm for sorting n distinct elements must make at least $\Omega(n \log n)$ comparisons in the worst case.

The clean way to formalize this is through a *decision tree*.

Decision-tree viewpoint

A deterministic comparison sort can be represented as a binary tree:

- each internal node asks a comparison such as "is $x_i < x_j$?";

- the two outgoing edges correspond to the two possible answers;
- each leaf contains the final output ordering.

If the algorithm always finishes within T comparisons, then the decision tree has height at most T, and therefore at most 2^T leaves.

On the other hand, if all inputs are distinct, there are $n!$ possible relative orders of the input elements. A correct sorting algorithm must be able to distinguish all of them. Equivalently, the decision tree must have at least $n!$ leaves.

Therefore $2^T \geq n! \approx (n/e)^n$, so $T \geq n \log_2(n/e)$.

Important takeaway. This is a lower bound for the *comparison model*, not for all possible algorithms.

4 Escaping the Comparison Model

The comparison lower bound disappears if the algorithm is allowed to use more information than just pairwise comparisons.

Counting sort

Suppose the keys are integers in $\{0, 1, \ldots, U\}$ – you can even think $U = n$ for simplicity. Then we can:

1. count how many times each value appears;
2. sweep through the counts in increasing order;
3. output each value the appropriate number of times.

This takes $O(n + U)$ time. So if $U = O(n)$, we get a linear-time sorting algorithm. This is impossible in the comparison model, but perfectly possible in word-RAM because the algorithm is exploiting the actual numeric values of the keys.

Radix sort for integers

Now suppose we have n integers, each at most n^{100}. In word-RAM, each such integer fits in one machine word.

Write each number in base n. Because each number is at most n^{100}, they have at most 100 base-n digits.

Now perform a stable counting sort on the least significant digit, then the next digit, and so on. There are only 100 passes, and each pass costs $O(n)$ because each digit lies in $\{0, 1, \ldots, n-1\}$. Therefore the total running time is $O(n)$.

A radix-sort example for integers

Suppose the input consists of n integers in the range

$$\{0, 1, \ldots, n^{100} - 1\}.$$

We will show that these integers can be sorted in $O(n)$ time in the word-RAM model.

Step 1: write numbers in base n. Every input integer x can be written uniquely as

$$x = a_0(x) + a_1(x)n + a_2(x)n^2 + \cdots + a_{99}(x)n^{99},$$

where each digit $a_t(x)$ lies in $\{0, 1, \ldots, n-1\}$. So each number has exactly 100 base-n digits.

Step 2: counting sort by one digit. Fix a digit position $t \in \{0, 1, \ldots, 99\}$. We describe how to stably sort the numbers by the digit $a_t(x)$ in time $O(n)$.

Since $a_t(x) \in \{0, 1, \ldots, n-1\}$, we can use an array

$$\texttt{count}[0], \texttt{count}[1], \ldots, \texttt{count}[n-1].$$

First compute, for each digit d,

$$\texttt{count}[d] = \#\{x : a_t(x) = d\}.$$

Then compute prefix sums

$$\texttt{start}[d] = \sum_{e<d} \texttt{count}[e],$$

so that $\texttt{start}[d]$ is the first output position reserved for numbers whose t-th digit equals d.

Now scan the input array from left to right. When we see an element x with digit $a_t(x) = d$, place it in the next free slot of the block for digit d, and increment that pointer. Because we scan from left to right, this sorting procedure is *stable*: if two elements have the same digit, then they appear in the output in the same relative order as in the input.

The running time is $O(n + n) = O(n)$.

Step 3: least-significant-digit radix sort. Now perform the following sequence of passes:

1. stably counting-sort by $a_0(x)$;
2. stably counting-sort by $a_1(x)$;
3. stably counting-sort by $a_2(x)$;
4. …
5. stably counting-sort by $a_{99}(x)$.

Running time. There are 100 passes, and each pass takes $O(n)$ time. Therefore the total running time is $O(100n) = O(n)$.

Why this does not contradict the lower bound. The comparison lower bound says that *comparison-based* sorting needs $\Omega(n \log n)$ comparisons. Radix sort is not comparison-based because it uses the digit representation of the numbers directly.

5 Radix Sort for Strings

Problem: Radix sort for words

Given English words $w_1, \ldots, w_n$, sort them alphabetically in time proportional to the total input length.

For simplicity we'll assume that all the w_i are length exactly L, so the target runtime is $O(nL)$.

Why comparison sorting is too slow

If we use comparison sorting, then:

- we need $\Theta(n \log n)$ comparisons;
- each comparison may inspect up to L characters.

So the total cost can be

$$\Theta(nL \log n),$$

which is worse than linear in the input size.

Radix sort

The algorithm is very similar to the previous section on sorting integers in base n. The idea is to sort by characters directly. Sort by the first letter – this costs $O(n)$ time because there are a constant number of letters in the alphabet. Split into groups based on matching first letter and recursively process each group. The number of layers of recursion is L, and the total size of groups at each layer is n. Processing a group takes linear time, so the total time is $O(nL)$.