To bound this, recall the integration formula for bounding summations (which we paraphrase here). For any monotonically increasing function $f(x)$

$$\sum_{i=a}^{b-1} f(i) \leq \int_a^b f(x)dx.$$

The function $f(x) = x \ln x$ is monotonically increasing, and so we have

$$S(n) \leq \int_2^n x \ln x dx.$$

If you are a calculus macho man, then you can integrate this by parts, and if you are a calculus wimp (like me) then you can look it up in a book of integrals

$$\int_2^n x \ln x dx = \frac{x^2}{2} \ln x - \frac{x^2}{4}\Big|_{x=2}^{n} = \left(\frac{n^2}{2} \ln n - \frac{n^2}{4}\right) - (2 \ln 2 - 1) \leq \frac{n^2}{2} \ln n - \frac{n^2}{4}.$$

This completes the summation bound, and hence the entire proof.

Summary: So even though the worst-case running time of QuickSort is $\Theta(n^2)$, the average-case running time is $\Theta(n \log n)$. Although we did not show it, it turns out that this doesn't just happen much of the time. For large values of n, the running time is $\Theta(n \log n)$ with high probability. In order to get $\Theta(n^2)$ time the algorithm must make poor choices for the pivot at virtually every step. Poor choices are rare, and so continuously making poor choices are very rare. You might ask, could we make QuickSort deterministic $\Theta(n \log n)$ by calling the selection algorithm to use the median as the pivot. The answer is that this would work, but the resulting algorithm would be so slow practically that no one would ever use it.

QuickSort (like MergeSort) is not formally an in-place sorting algorithm, because it does make use of a recursion stack. In MergeSort and in the expected case for QuickSort, the size of the stack is $O(\log n)$, so this is not really a problem.

QuickSort is the most popular algorithm for implementation because its actual performance (on typical modern architectures) is so good. The reason for this stems from the fact that (unlike Heapsort) which can make large jumps around in the array, the main work in QuickSort (in partitioning) spends most of its time accessing elements that are close to one another. The reason it tends to outperform MergeSort (which also has good locality of reference) is that most comparisons are made against the pivot element, which can be stored in a register. In MergeSort we are always comparing two array elements against each other. The most efficient versions of QuickSort uses the recursion for large subarrays, but once the sizes of the subarray falls below some minimum size (e.g. 20) it switches to a simple iterative algorithm, such as selection sort.

Lecture 16: Lower Bounds for Sorting

(Thursday, Mar 19, 1998)

Read: Chapt. 9 of CLR.

Review of Sorting: So far we have seen a number of algorithms for sorting a list of numbers in ascending order. Recall that an *in-place* sorting algorithm is one that uses no additional array storage (however, we allow QuickSort to be called in-place even though they need a stack of size $O(\log n)$ for keeping track of the recursion). A sorting algorithm is *stable* if duplicate elements remain in the same relative position after sorting.

Slow Algorithms: Include BubbleSort, InsertionSort, and SelectionSort. These are all simple $\Theta(n^2)$ in-place sorting algorithms. BubbleSort and InsertionSort can be implemented as stable algorithms, but SelectionSort cannot (without significant modifications).

Mergesort: Mergesort is a stable $\Theta(n \log n)$ sorting algorithm. The downside is that MergeSort is the only algorithm of the three that requires additional array storage, implying that it is not an in-place algorithm.

Quicksort: Widely regarded as the *fastest* of the fast algorithms. This algorithm is $O(n \log n)$ in the *expected case*, and $O(n^2)$ in the worst case. The probability that the algorithm takes asymptotically longer (assuming that the pivot is chosen randomly) is extremely small for large n. It is an (almost) in-place sorting algorithm but is not stable.

Heapsort: Heapsort is based on a nice data structure, called a *heap*, which is a fast priority queue. Elements can be inserted into a heap in $O(\log n)$ time, and the largest item can be extracted in $O(\log n)$ time. (It is also easy to set up a heap for extracting the smallest item.) If you only want to extract the k largest values, a heap can allow you to do this is $O(n + k \log n)$ time. It is an in-place algorithm, but it is not stable.

Lower Bounds for Comparison-Based Sorting: Can we sort faster than $O(n \log n)$ time? We will give an argument that if the sorting algorithm is based solely on making comparisons between the keys in the array, then it is impossible to sort more efficiently than $\Omega(n \log n)$ time. Such an algorithm is called a *comparison-based* sorting algorithm, and includes all of the algorithms given above.

Virtually all known general purpose sorting algorithms are based on making comparisons, so this is not a very restrictive assumption. This does not preclude the possibility of a sorting algorithm whose actions are determined by other types of operations, for example, consulting the individual bits of numbers, performing arithmetic operations, indexing into an array based on arithmetic operations on keys.

We will show that any *comparison-based* sorting algorithm for a input sequence $\langle a_1, a_2, \ldots, a_n \rangle$ must make at least $\Omega(n \log n)$ comparisons in the worst-case. This is still a difficult task if you think about it. It is easy to show that a problem *can* be solved fast (just give an algorithm). But to show that a problem *cannot* be solved fast you need to reason in some way about all the possible algorithms that might ever be written. In fact, it seems surprising that you could even hope to prove such a thing. The catch here is that we are limited to using comparison-based algorithms, and there is a clean mathematical way of characterizing all such algorithms.

Decision Tree Argument: In order to prove lower bounds, we need an abstract way of modeling "any possible" comparison-based sorting algorithm, we model such algorithms in terms of an abstract model called a *decision tree*.

In a *comparison-based* sorting algorithm only comparisons between the keys are used to determine the action of the algorithm. Let $\langle a_1, a_2, \ldots, a_n \rangle$ be the input sequence. Given two elements, a_i and a_j, their relative order can only be determined by the results of comparisons like $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, and $a_i > a_j$.

A decision tree is a mathematical representation of a sorting algorithm (for a fixed value of n). Each node of the decision tree represents a comparison made in the algorithm (e.g., $a_4 : a_7$), and the two branches represent the possible results, for example, the left subtree consists of the remaining comparisons made under the assumption that $a_4 \leq a_7$ and the right subtree for $a_4 > a_7$. (Alternatively, one might be labeled with $a_4 < a_7$ and the other with $a_4 \geq a_7$.)

Observe that once we know the value of n, then the "action" of the sorting algorithm is completely determined by the results of its comparisons. This action may involve moving elements around in the array, copying them to other locations in memory, performing various arithmetic operations on non-key data. But the bottom-line is that at the end of the algorithm, the keys will be permuted in the final array

in some way. Each leaf in the decision tree is labeled with the final permutation that the algorithm generates after making all of its comparisons.

To make this more concrete, let us assume that $n = 3$, and let's build a decision tree for SelectionSort. Recall that the algorithm consists of two phases. The first finds the smallest element of the entire list, and swaps it with the first element. The second finds the smaller of the remaining two items, and swaps it with the second element. Here is the decision tree (in outline form). The first comparison is between a_1 and a_2. The possible results are:

$a_1 \leq a_2$**:** Then a_1 is the current minimum. Next we compare a_1 with a_3 whose results might be either:

- $a_1 \leq a_3$**:** Then we know that a_1 is the minimum overall, and the elements remain in their original positions. Then we pass to phase 2 and compare a_2 with a_3. The possible results are:
  - $a_2 \leq a_3$**:** Final output is $\langle a_1, a_2, a_3 \rangle$.
  - $a_2 > a_3$**:** These two are swapped and the final output is $\langle a_1, a_3, a_2 \rangle$.
- $a_1 > a_3$**:** Then we know that a_3 is the minimum is the overall minimum, and it is swapped with a_1. The we pass to phase 2 and compare a_2 with a_1 (which is now in the third position of the array) yielding either:
  - $a_2 \leq a_1$**:** Final output is $\langle a_3, a_2, a_1 \rangle$.
  - $a_2 > a_1$**:** These two are swapped and the final output is $\langle a_3, a_1, a_2 \rangle$.

$a_1 > a_2$**:** Then a_2 is the current minimum. Next we compare a_2 with a_3 whose results might be either:

- $a_2 \leq a_3$**:** Then we know that a_2 is the minimum overall. We swap a_2 with a_1, and then pass to phase 2, and compare the remaining items a_1 and a_3. The possible results are:
  - $a_1 \leq a_3$**:** Final output is $\langle a_2, a_1, a_3 \rangle$.
  - $a_1 > a_3$**:** These two are swapped and the final output is $\langle a_2, a_3, a_1 \rangle$.
- $a_2 > a_3$**:** Then we know that a_3 is the minimum is the overall minimum, and it is swapped with a_1. We pass to phase 2 and compare a_2 with a_1 (which is now in the third position of the array) yielding either:
  - $a_2 \leq a_1$**:** Final output is $\langle a_3, a_2, a_1 \rangle$.
  - $a_2 > a_1$**:** These two are swapped and the final output is $\langle a_3, a_1, a_2 \rangle$.

The final decision tree is shown below. Note that there are some nodes that are unreachable. For example, in order to reach the fourth leaf from the left it must be that $a_1 \leq a_2$ and $a_1 > a_2$, which cannot both be true. Can you explain this? (The answer is that virtually all sorting algorithms, especially inefficient ones like selection sort, may make comparisons that are redundant, in the sense that their outcome has already been determined by earlier comparisons.) As you can see, converting a complex sorting algorithm like HeapSort into a decision tree for a large value of n will be very tedious and complex, but I hope you are convinced by this exercise that it can be done in a simple mechanical way.

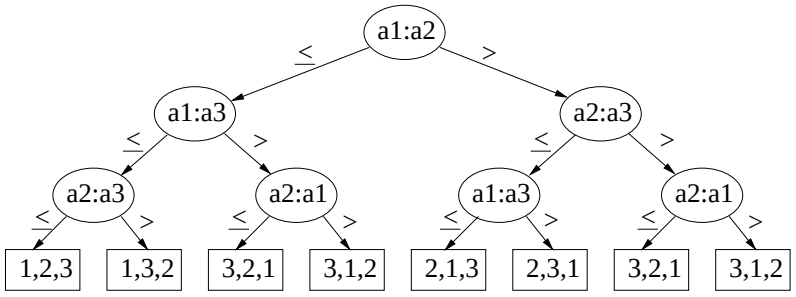

Figure 16: Decision Tree for SelectionSort on 3 keys.

Using Decision Trees for Analyzing Sorting: Consider any sorting algorithm. Let $T(n)$ be the maximum number of comparisons that this algorithm makes on any input of size n. Notice that the running time fo the algorithm must be at least as large as $T(n)$, since we are not counting data movement or other computations at all. The algorithm defines a decision tree. Observe that the height of the decision tree is exactly equal to $T(n)$, because any path from the root to a leaf corresponds to a sequence of comparisons made by the algorithm.

As we have seen earlier, any binary tree of height $T(n)$ has at most $2^{T(n)}$ leaves. This means that this sorting algorithm can *distinguish* between at most $2^{T(n)}$ different final actions. Let's call this quantity $A(n)$, for the number of different final actions the algorithm can take. Each action can be thought of as a specific way of permuting the oringinal input to get the sorted output.

How many possible actions must any sorting algorithm distinguish between? If the input consists of n distinct numbers, then those numbers could be presented in any of $n!$ different permutations. For each different permutation, the algorithm must "unscramble" the numbers in an essentially different way, that is it must take a different action, implying that $A(n) \geq n!$. (Again, $A(n)$ is usually not exactly equal to $n!$ because most algorithms contain some redundant unreachable leaves.)

Since $A(n) \leq 2^{T(n)}$ we have $2^{T(n)} \geq n!$, implying that

$$T(n) \geq \lg(n!).$$

We can use *Stirling's approximation* for $n!$ (see page 35 in CLR) yielding:

$$\begin{aligned} n! &\geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \\ T(n) &\geq \lg\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \\ &= \lg\sqrt{2\pi n} + n\lg n - n\lg e \ \in \ \Omega(n\log n). \end{aligned}$$

Thus we have, the following theorem.

Theorem: Any comparison-based sorting algorithm has worst-case running time $\Omega(n \log n)$.

This can be generalized to show that the *average-case* time to sort is also $\Omega(n \log n)$ (by arguing about the average height of a leaf in a tree with at least $n!$ leaves). The lower bound on sorting can be generalized to provide lower bounds to a number of other problems as well.

Lecture 17: Linear Time Sorting

(Tuesday, Mar 31, 1998)

Read: Chapt. 9 of CLR.

Linear Time Sorting: Last time we presented a proof that it is not possible to sort faster than $\Omega(n \log n)$ time assuming that the algorithm is based on making 2-way comparisons. Recall that the argument was based on showing that any comparison-based sorting could be represented as a decision tree, the decision tree must have at least $n!$ leaves, to distinguish between the $n!$ different permutations in which the keys could be input, and hence its height must be at least $\lg(n!) \in \Omega(n \lg n)$.

This lower bound implies that if we hope to sort numbers faster than in $O(n \log n)$ time, we cannot do it by making comparisons alone. Today we consider the question of whether it is possible to sort without the use of comparisons. They answer is yes, but only under very restrictive circumstances.