CS 624
Lecture 5: A Little More About Sorting

1 How well can we really do?

We have seen that naive sorting algorithms such as insertion sort and selection sort are $O(n^2)$ on average. Quicksort does a lot better—it is $O(n \log n)$ on average, but in the worst case it is still $O(n^2)$. Heapsort and mergesort are guaranteed to be $O(n \log n)$ in the worst case, so in some sense they are more reliable than quicksort. In general, however, quicksort runs considerably faster than either of them, partly because the computations in the inner loop are just simpler, and in part because quicksort exhibits more locality of reference. So some version of quicksort (typically with randomization) is generally the method of choice, although there are exceptions.

There is an obvious question here: Can we do even better? Is there a method of sorting which is guaranteed to run in time $O(n)$, for instance? That is, is there a method of sorting whose worst-case running time is $O(n)$?

2 A lower bound for comparison sorting

The answer turns out to be "maybe". But first we can show that for a very large class of sorting algorithms—a class that includes all the algorithms we have considered so far—the worst-case running time really is $\Omega(n \log n)$.

We will consider the class of algorithms whose processing can be modeled by a binary decision tree. That is, the algorithm can be considered to consist of a sequence of steps. At each step a choice is made between two possibilities. The choice that is made determines the next choice that is made, and so on.

All the sorting algorithms we have seen so far work by comparing elements with each other. These are called *comparison sorts.* All comparison sorts can be modeled by a binary decision tree. For example, let's consider an insertion sort operating on three elements $\{a_1, a_2, a_3\}$. The binary decision tree for this algorithm is shown in Figure 1.

The binary decision tree for quicksort is a bit more complicated. Figure 2 shows the top part of the tree, which handles the first partition step.

2.1 Theorem *In a sorting algorithm modeled by a binary decision tree, the worst-case running time is $\Omega(n \log n)$.*

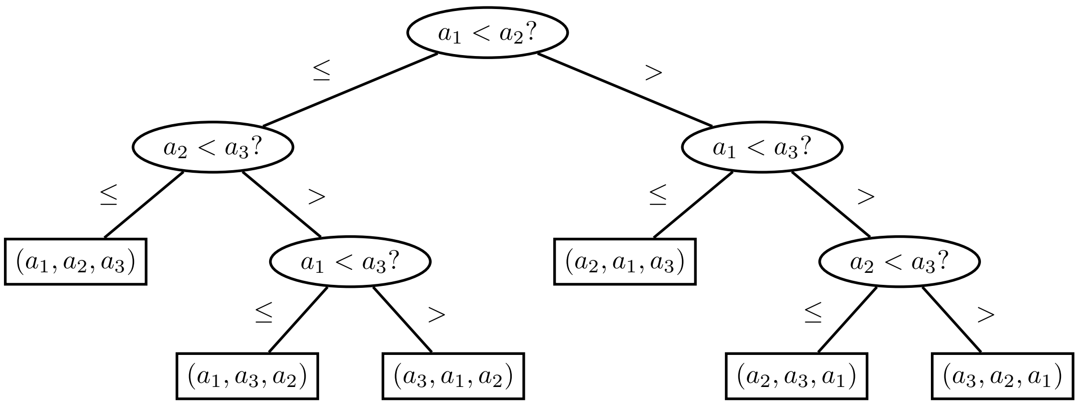

Figure 1: The binary decision tree for an insertion sort on three elements. The symbol a_i refers to the value that was originally in position i at the start of the algorithm. It may be moved to one or more other positions as the algorithm executes.

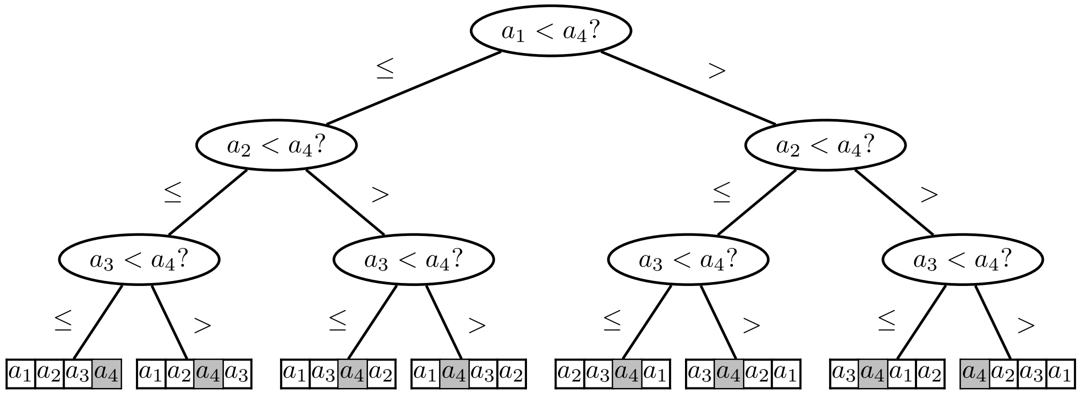

Figure 2: The binary decision tree for the initial partition step of quicksort on four elements. As before, a_i refers to the value that was originally in position i at the start of the algorithm. It may be moved to one or more other positions as the algorithm executes.

PROOF. The worst-case running time is bounded below by the depth of the decision tree. The number of leaves in the decision tree must be the number of possible permutations, which is $n!$. The depth of a binary tree with L leaves is $\Omega(\log L)$. Therefore the depth of the decision tree is

$$\Omega(\log n!) = \Omega(n \log n)$$ □

2.2 Exercise *Prove that the depth of a binary tree with L leaves is $\Omega(\log L)$.*

3 Bucketsort

Nevertheless there are sorting algorithms that can sometimes be used and yield better than $\Omega(n \log n)$ running time.

For instance, one nice kind of sorting is called *bucketsort*. Here's the simplest case: suppose we have the set of integers $1\,..\,n$ *in some order*—say they come to us as the sequence $\{a_1, a_2, \dots, a_n\}$, and we want to put them in the "right order", i.e., we want to sort them. Well this is easy: we just allocate an array $A[1\,..\,n]$ and put each integer a_k in $A[a_k]$. This just takes time $O(n)$.

Of course this is a pretty artificial case, but it contains a useful idea. Suppose to make things a little more general that again we have a set of n integers $\{a_1, a_2, \dots, a_n\}$, but now each integer can be in the range $1 \le a_k \le M$. (In other words, we are allowing M to be greater than n.)

Again we can create an array $A[1\,..\,n]$. The elements of this array will be sets of integers. We call each of these sets a "bucket", and initially, each bucket will be empty. We will put each a_k in its appropriate set. The array is shown in Figure 3. As you can see from the picture, the bucket $A[j]$ will contain numbers a_k such that

$$\left\lfloor \frac{(j-1)M}{n} \right\rfloor + 1 \le a_k \le \left\lfloor \frac{jM}{n} \right\rfloor$$

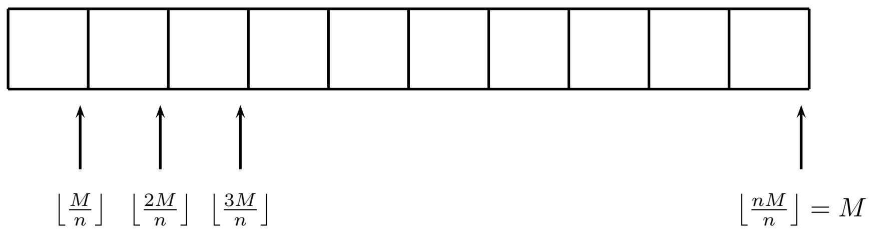
$\lfloor \frac{M}{n} \rfloor$
$\lfloor \frac{2M}{n} \rfloor$
$\lfloor \frac{3M}{n} \rfloor$
$\lfloor \frac{nM}{n} \rfloor = M$

Figure 3: Bucket sorting n numbers in the range $1\,..\,M$. The largest number held in each bucket is shown below it.

3.1 Exercise *Each number a_k will go in some bucket $A[j]$. The index j depends on the value a_k, and it also depends on M and n. That is, j is a function of a_k, M, and n. Find a formula for this function. That is, find a formula involving a_k, M, and n whose value is the index j of the bucket $A[j]$ into which a_k should be placed. (The formula is actually quite simple.)*

So we put all the numbers $\{a_k : 1 \le k \le n\}$ in the buckets. Then what happens?

There are two possibilities:

1. In the best possible case, there is exactly 1 number in each bucket. In that case, we are effectively done—we just walk through the buckets and output the elements in the order in which we find them.

2. However, it is entirely possible that some buckets have more than 1 element in them, and others are empty. In that case, we need to sort the elements in each bucket and then walk through the buckets in order and output the elements as we find them.

So what is the cost of doing this in general, i.e., if there may be more than 1 element in a bucket?

Here's what Skiena says in his course:

> Certainly the average number of elements in a bucket is 1. So on average, sorting the numbers in each bucket takes $O(1)$ time. Then we just have to concatenate all the sorted buckets. In this way we see that the whole operation takes $O(n)$ time.

This is "morally correct"—that is, it is more or less intuitively correct—but actually the reasoning is wrong. The reason it is wrong is that the cost of sorting the elements in each bucket is almost certainly not proportional to the number of elements in the bucket. Suppose for instance, we use insertion sort to sort the elements in each bucket. (Remember that insertion sort is not a bad sort routine to use for small numbers of elements, and we're pretty sure that the number of elements in each bucket is small.) Well, if bucket i has n_i elements in it, then insertion sort takes (on average) $O(n_i^2)$ time. What Skiena pointed out is that the average value of n_i is 1. This is true, because certainly $\sum_{i=1}^{n} n_i = n$,and so $\frac{1}{n}\sum_{i=0}^{n} n_i = 1$. But the average cost of sorting each bucket will not be the average value of n_i—it will be the average value of n_i^2. And this can be different.

For instance, the average of the two numbers 0 and 2 is 1. But the average of their squares is 2.

So we really need to find the average value of n_i^2.

The text has a very neat way of doing this. You may find it just perfect. It involves some notions of probability theory that you may not be familiar with. So I'm going to present another way of doing it here, based on generating functions. Either way of course is fine. (And best of all, they both give the same answer!)

First of all, the average value of n_i^2 must be the same as the average value of n_j^2 for any i and j. In other words, it doesn't matter which bucket we pick—the average value of the square of the number of elements in that bucket will be the same. This is just by symmetry: if every element is picked randomly and independently from all the buckets, there is no way that any bucket would be favored over any other. So we can restrict ourselves to computing n_1^2, say. Now

$$\text{the average value of } n_1^2 = \sum_{j=0}^{n} (\text{the probability that exactly } j \text{ numbers land in bucket 1}) \cdot j^2$$

(Actually, the preferred way of saying "the average value of n_1^2" is "the expected value of n_1^2". But they really mean the same thing.)

Now since we are considering random choices of numbers, the probability that exactly j numbers land in bucket 1 is the probability that j land in bucket 1 *and* $(n-j)$ *don't.* The number of ways of picking j out of the n numbers to land in bucket 1 is just $\binom{n}{j}$. The probability that any particular one of those ways happens is just $\left(\frac{1}{n}\right)^j\left(1-\frac{1}{n}\right)^{n-j}$ Therefore the probability that exactly j numbers land in bucket 1 is

$$\Big(\frac{1}{n}\Big)^j\Big(1-\frac{1}{n}\Big)^{n-j}\binom{n}{j}$$

and so the expected value of n_1^2 is

$$\sum_{j=0}^{n}\Big(\frac{1}{n}\Big)^j\Big(1-\frac{1}{n}\Big)^{n-j}\binom{n}{j}j^2$$

To evaluate this, we will use a generating function: we figure that the factor j^2 (at the end, on the right) can be arrived at by repeated differentiation, and the rest looks a lot like the binomial theorem. So let us set

$$f(x)=\sum_{j=0}^{n}\Big(\frac{1}{n}\Big)^j\Big(1-\frac{1}{n}\Big)^{n-j}\binom{n}{j}x^j$$

Then we have

$$f'(x)=\sum_{j=0}^{n}\Big(\frac{1}{n}\Big)^j\Big(1-\frac{1}{n}\Big)^{n-j}\binom{n}{j}jx^{j-1}$$

Now we can't just differentiate again, because we would get $j(j-1)$. We want j^2. So we multiply by x first:

$$xf'(x)=\sum_{j=0}^{n}\Big(\frac{1}{n}\Big)^j\Big(1-\frac{1}{n}\Big)^{n-j}\binom{n}{j}jx^j$$

and then we can differentiate:

$$\big(xf'(x)\big)'=\sum_{j=0}^{n}\Big(\frac{1}{n}\Big)^j\Big(1-\frac{1}{n}\Big)^{n-j}\binom{n}{j}j^2x^{j-1}$$

So let us set $g(x)=\big(xf'(x)\big)'$. Then we see that the expected value of n_1^2 is just $g(1)$. So if we can find a nice expression for g, we will be done. So let's go back and look at f: We have

$$\begin{aligned}f(x)&=\sum_{j=0}^{n}\Big(\frac{1}{n}\Big)^j\Big(1-\frac{1}{n}\Big)^{n-j}\binom{n}{j}x^j\\&=\sum_{j=0}^{n}\Big(\frac{x}{n}\Big)^j\Big(1-\frac{1}{n}\Big)^{n-j}\binom{n}{j}\qquad\text{Now apply the binomial theorem!}\\&=\Big(1+\frac{x-1}{n}\Big)^n\end{aligned}$$

That's not a bad expression. So now we can figure out what g is:

$$f'(x)=n\Big(1+\frac{x-1}{n}\Big)^{n-1}\cdot\frac{1}{n}=\Big(1+\frac{x-1}{n}\Big)^{n-1}$$

and then

$$\begin{aligned} g(x) &= \bigl(xf'(x)\bigr)' \\ &= \biggl(x\Bigl(1+\frac{x-1}{n}\Bigr)^{n-1}\biggr)' \\ &= \Bigl(1+\frac{x-1}{n}\Bigr)^{n-1} + (n-1)x\Bigl(1+\frac{x-1}{n}\Bigr)^{n-2}\cdot\frac{1}{n} \\ &= \Bigl(1+\frac{x-1}{n}\Bigr)^{n-1} + \Bigl(1-\frac{1}{n}\Bigr)x\Bigl(1+\frac{x-1}{n}\Bigr)^{n-2} \end{aligned}$$

and it is easy to see that by substituting 1 for x, we get

$$g(1) = 1 + \Bigl(1-\frac{1}{n}\Bigr) = 2 - \frac{1}{n}$$

and so that is the expected value of n_1^2, and in fact is the expected value of n_i^2 for any i.

So even though Skiena's reasoning was slightly incorrect, his conclusion was correct: the average time to sort each bucket is $O(1)$, and there are n buckets, so the average time for the entire sort is $O(n)$.

4 So why don't we always use bucketsort?

What happened to our $\Omega(n \log n)$ lower bound for the running time?

You can look at this from several different points of view.

- In the simplest case, where we are sorting the n integers from 1 to n, the method we are using is not modeled by a binary decision tree. So the $\Omega(n \log n)$ bound does not apply.

  We are able to do this because we have some very significant additional information about our input values. That is, in this case, each input value is known to go into a separate bucket; and once they do, we're done. In general, we don't have this kind of information—if all we knew was that we had n real numbers, they all might go into the same bucket, for all we know.

- In the general case, we didn't compute the worst-case cost at all; we computed the *average-case* cost. And the average case really depends for its success on the elements being distributed reasonably uniformly, so that each bucket gets only a very small number of elements.

  The worst-case, however, would occur when all the elements wind up in one bucket. In that case, we have just wasted some time—we still have to go ahead and sort that bucket. So we haven't gained anything at all.

In other words, bucketsort really only works when we can be reasonably sure that the elements are distributed fairly uniformly. And often in sorting problems we really don't know much at all about our input data. Skiena has a really amusing example of this; you should definitely read it.