601.433/633 Introduction to Algorithms **Lecturer:** Michael Dinitz
Topic: Sorting lower bound and Linear-time sorting **Date:** 9/10/24

5.1 Introduction

You should all know a few ways of sorting in $O(n \log n)$ time, most notably mergesort and the version of quicksort that we talked about at the end of last lecture, where we choose a pivot using the median-of-medians deterministic median-finding algorithm (we've also seen that randomized quicksort achieves this bound in expectation). The fact that this bound keeps popping up leads to a natural question: is it tight? Is there any hope of giving an $O(n)$-time algorithm, or even just an $o(n \log n)$-time algorithm? We'll see in this lecture that the answer is both yes and no: there are no sorting algorithms that run in $o(n \log n)$ time in the worst case that work like all of the sorting algorithms we've seen so far (i.e., that compare elements), but if we assume some extra structure in the elements (e.g., if they are integers) then in fact we can sort in linear time.

So far all of the sorting algorithm we've seen work in the *comparison model*: other than rearranging or moving elements around, all that they do is compare elements. Slightly more formally, we assume that we are given a (constant-time) algorithm that when given two elements a, b can test if $a < b$, but this is the only access that we are given to the elements. This makes algorithms in the comparison model (such as mergesort or quicksort) extremely general, as they can be used to sort arbitrary ordered sets where the elements have no extra structure.

On the other hand, lots of things that we sort (e.g., integers or strings) have extra structure. If you give me two integers a and b, not only can I test if $a < b$, I can also perform tests like "is a == 5?" or "is the third digit of a larger than the third digit of b?". We cannot do tests like this in the comparison model, as there is no notion of digit or absolute constants such as 5.

5.2 Sorting lower bound

We will now prove one of the most famous lower bounds in algorithms: no algorithm can sort in time $o(n \log n)$ in the comparison model. We will in fact prove the following slightly stronger theorem:

Theorem 5.2.1 *Any sorting algorithm in the comparison model must make at least* $\log(n!) = \Theta(n \log n)$ *comparisons in the worst case.*

This theorem is a lower bound on the number of comparisons, so we are essentially letting the algorithm reorder elements, move them around, copy them, etc. for free.

There are a few ways to prove this, which are essentially all equivalent. I'll present a method based on decision trees, but there are also other ways of seeing it.

The thing that makes this bound hard is that we want the bound to hold for *all* algorithms. So we can't just assume that the algorithm works by selecting a pivot (like quicksort) or sorting subarrays recursively (like mergesort). Instead, we need to find a way of reasoning about all possible algorithms.

The trick to doing this is to think of the input arrays not as arrays, but as *permutations*. For example, suppose we are given the unsorted array $A = [23, 14, 2, 5, 76]$. This corresponds to the permutation $(3, 2, 0, 1, 4)$, i.e., the permutation π where $\pi(0) = 3, \pi(1) = 2, \pi(2) = 0, \pi(3) = 1, \pi(4) = 4$. In other words, $A[i]$ is the $\pi(i)$'th smallest element: $A[0]$ is the third smallest element, $A[1]$ is the second smallest, $A[2]$ is the 0'th smallest (i.e., the smallest), $A[3]$ is the first smallest, and $A[4]$ is the 4th smallest. Note that there is exactly one permutation corresponding to each input.

Suppose that we knew this permutation (call it π); then we could easily construct the correct sorted array without any more comparisons by simply scanning through A and setting $B[\pi(i)] = A[i]$. On the other hand, if we could sort A into some array B, then we could construct π by setting $\pi(i)$ to the location in B which contains $A[i]$. This can be done with no extra comparisons if we simply "tag" each element in the input array with its starting location, so the element in $B[i]$ contains not just the element itself but also its initial location in A. For example, instead of sorting A we would instead sort $A' = [(23, 0), (14, 1), (2, 2), (5, 3), (76, 4)]$ into $B' = [(2, 2), (5, 3), (14, 1), (23, 0), (76, 4)]$. Then we could construct π by iterating from $i = 0$ to $n-1$ and letting $\pi(B'[i, 1]) = i$ (where $B'[i, 1]$ denotes the second element, i.e. the tag, of the i'th pair in B').

This second fact, that given a sorting algorithm we can construct π without doing any more comparisons between elements, implies that if we can prove that any algorithm for *finding* π takes $f(n)$ comparisons, then any algorithm for sorting must also take $f(n)$ comparisons (since if there were a sorting algorithm with $g(n) < f(n)$ comparisons then we would also have a computing π algorithm which only takes $g(n) < f(n)$ comparisons).

So instead of trying to "sort", let's think of trying to find a specific (unknown) permutation of n elements, where the only tool we have available is comparisons that correspond to this permutation. By the above discussion, this is the exact same problem as sorting: if we can do one of them in x comparisons, then we can also do the other in x comparisons. So let's think of trying to prove lower bounds for this "finding a hidden permutation" problem.

Suppose that we do a comparison of a and b. If $a < b$ then some permutations are not possible (those which put a after b), while some are. Similarly, if $a > b$ then some permutations are not possible while some are. Informally, this means that we can think of each comparison as cutting down the space of possible permutations. We start with all permutations, do some comparison after which some subset of permutations is still possible, then do another comparison which further restricts the set of possible permutations, etc. Note that the comparison we choose to make can depend on the outcome of the previous comparisons (i.e. our algorithm can be *adaptive*).

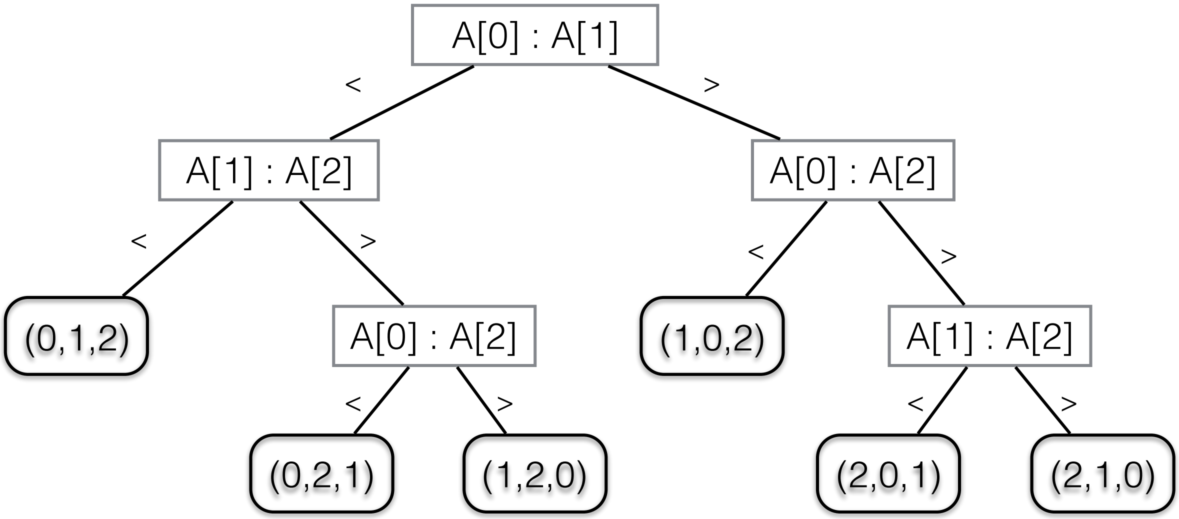
It is not hard to see that all of the previous sorting algorithms we have discussed can be put into this model. Take quicksort, for example. We choose a pivot, and the next $n - 1$ comparisons are just comparing each element to the pivot. We then recurse on L, picking a pivot and comparing all elements in L to that pivot, etc. Once we return from the recursive call to L (having done some collection of comparisons), we pick a pivot for G and compare everything to that, and continuing down the recursion. At the end, we will have done some set of comparisons, the outcome of which will uniquely determine the correct permutation.

A very natural way to model this kind of algorithm is as a *decision tree*. Each node in the tree corresponds to a comparison, and there are two children corresponding to the outcome of the comparison. Any sorting algorithm cannot stop when there are still multiple permutations consistent with the comparisons it has made so far, and thus each leaf of this tree is a single permutation which must be uniquely identified by the outcomes corresponding to the path from it up to the root.

Let's do a small example, with $n = 3$ and an input array A. Then there are six possible input orderings:

$$(0,1,2), \quad (0,2,3), \quad (1,0,2), \quad (1,2,0), \quad (2,0,1), \quad (2,1,0)$$

Suppose our algorithm first tests whether $A[0] < A[1]$. If it is, then there are still three possible input orderings $((0,1,2),(0,2,1),(1,2,0))$. If it is not, then there are also still three possible input orderings. Depending on the outcome, we will do more comparisons in order to narrow down the possibilities until we can eventually figure out what the input permutation actually is (an informal way to think about this is as a game of "20 questions"). So our sorting algorithm might correspond to a decision tree which looks like:



Given a tree, the number of comparisons it makes in the worst case is simply its depth, i.e. the maximum over all leaves of their distance from the root. What is the minimum depth possible? Since each leaf corresponds to a unique permutation there must be $n!$ leaves, and since the decision tree is binary this means the depth must be at least $\log(n!) = \Theta(n \log n)$. That's the sorting lower bound!

5.3 Linear-time sorting

If the sorting lower bound implies that any sorting algorithm must make $\Omega(n \log n)$ comparisons, how can we hope for a linear-time algorithm? The trick is to leave the comparison model, and give ourselves access to the elements beyond simple comparisons. For the rest of today, we will talk about integers, but most of this can be applied to strings or other data that has some simple representation.

5.3.1 Counting sort / Bucket sort

We've generally been assuming that all elements are distinct. What if we stop making this assumption? Moreover, what if in fact almost the opposite is true: all values are within a very small range? For example, suppose we are sorting n integers, but they all take one of k values (e.g. maybe they all lie in $\{1, \dots, k\}$). Then we can simply maintain a count for each possible value of the number of times it appears in the array. We then simply go through the values from smallest to largest, outputting the correct number of copies.

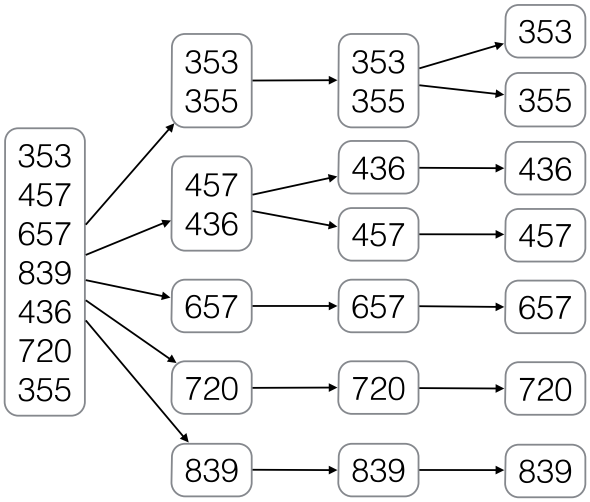
The running time of this algorithm is clearly $O(n + k)$: we need to initialize an array of length k to be all 0's, and then seeing each element and increasing the count takes time $O(n)$, and going through the count array to output the right values also takes time $O(n)$. One note about something which will be important later: it looks like we can just get rid of the k, but we can't. If k is larger than n then you might try to just maintain a hash table or something similar to only need to store the (at most) n values which have nonzero count, but the fact that we need to scan this in the end from smallest to largest means we have to use an associative array data structure which support an operation like findMin(), i.e. a priority queue. This makes some operation (either inserts or deletes) take $\Omega(\log n)$ time, which we're trying to avoid.

Another important feature is that we can make this algorithm *stable*: if two elements are tied, then their ordering is preserved after sorting. In other words, if $a = b$ and a comes before b in the unsorted array, then a will also come before b in the sorted array. As defined, counting sort just keeps counts of the values rather than storing the objects with those values (which might be what we want if we're sorting longer records), but it is easy to have each entry in the array be (say) a pointer to the first element in a linked list of the items with that same value. By inserting at the end of the list (which we can still do in constant time by also keeping pointers to the end of the list and the element before the end), when we iterate through to create the sorted array we preserve the relative orders of tied items.

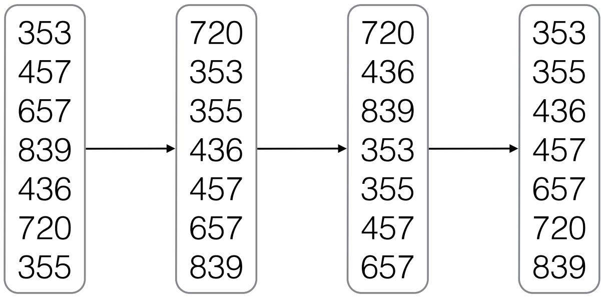
5.3.2 Radix Sort

Counting sort is great when k is small. But what if k is larger than n? For example, what if $k = n^3$? We will design a different algorithm, radix sort, which will use counting sort as a black box but will sort in $O(n)$ time (with a few caveats).

Let's start somewhat concretely. Suppose that we represent each integer using its decimal (i.e. base 10) expansion, and let's assume that all integers have the same number of digits (i.e. small integers are padded out with 0s to have the same number of digits). One obvious thing we could do is put the elements in buckets by their most significant digit (i.e. do bucket sort on the most significant digit), recursively sort each bucket where we cut off the most significant digit, and then output the buckets in order from the largest most significant digit to the smallest. In other words, we do bucket sort on the digits from most significant to least significant, each time restricting ourselves to the bucket under consideration. This algorithm is called *radix sort*, since we operate one digit at a time. Let's do an example:



This algorithm works, but is somewhat clunky. For example, we need to use up a lot of memory to partition the items into buckets, since we always have to remember the full set of buckets (to make sure that we only run the sort inside of each bucket). Somewhat surprisingly, we can actually sort without buckets by sorting from the *least* significant digit to the *most*! That is, we first sort based on the last digit, then sort based on the next-to-last digit, then the third-to-last, etc. As long as the sort we use is stable (as is the version of bucket sort we discussed), this will end up giving the correct sorted array.



Theorem 5.3.1 *Radix sort from least significant to most significant is correct if the sort used on each digit is stable.*

Proof: We can use induction on the number of digits we have sorted. After we sort the least significant digits, the items are correctly sorted by their least significant digit (by definition). Now suppose that the elements are correctly sorted based on their last i digits. In other words, for each number we look at the at the last i digits and interpret them as a number between 0 and $10^i - 1$, and we assume that the (full) numbers are sorted correctly based just on these. Then when we sort the $(i+1)$st least significant digit, the values with different $(i+1)$st least significant digits become correctly sorted, and values with the same $(i+1)$st least significant digit become correctly sorted because we use a stable sort and by induction they were already correctly sorted based on their last i digits. So the induction holds, and at the end of the process all values are correctly sorted. ■

Note that by sorting from least significant digit to most we avoided all of the complications of bucketing. This doesn't matter much asymptotically, but is nice for the constants and makes it easier to implement. It's also just a really slick algorithm, which feels a lot less hack-y.

Running time: The most interesting thing about radix sort is its running time. It's not too hard to analyze the version we came up with, particularly the least-significant first version. If there are d digits, we do d bucket sorts, and each digit can take one of 10 values. So the total time is $d(n+k) = d(n+10)$. This doesn't seem like much of an improvement since $d \geq \log_{10} n$ (if all integers are distinct), so the running time is still $\Theta(n \log n)$. However, there is a sense in which we can already see an improvement: if every number one of the n numbers uses $\Omega(d)$ bits, then we could think of the "size" of the input as $N = \Omega(dn)$, in which case we are already running in time linear in the "size" of the input.

But can we achieve running time of $O(n)$, not just $O(dn)$? In some sense this requires "cheating", since if it takes one unit of time to look at one digit, then just looking at the input already takes $\Omega(dn)$ time. But what if I give you the ability not just to look at one digit at time, but to actually look at "groups" of digits in one unit of time? To instantiate this, what if we consider b digits at a time. In other words, we first bucket sort based on the last b digits (considered as a number between 0 and $10^b - 1$), then the next b digits, etc. Then in each bucket sort there are $k = 10^b$ possible values, and we have to do d/b bucket sorts. Thus the total number of comparisons becomes

$$\frac{d}{b}\left(n + 10^b\right)$$

This is great! If the number of digits isn't too much larger than $\log n$, we can plug in $b = \log n$ and we get a linear time sorting algorithm. Slightly more formally, if $d \leq O(\log n)$ then if we set $b = \log_{10} n$ the number of comparisons becomes $O(n)$.

So, for example, suppose that we are trying to sort a collection of integers which we know are between 0 and n^9. Then they should have at most $9 \log_{10} n = O(\log n)$ digits, so if we use $b = \log_{10} n$ the total number of comparisons is at most $9(n+n) = 18n$.

A few notes about this. First, I've done this with digits and base 10, but clearly on a real computer it's more efficient to use bits and base 2. Base 10 is traditionally used when we think of radix sort (but not when we implement it) mostly for historical reasons. Second, we clearly "cheated" somewhat – we are assuming that it costs 1 comparison to compare bits, but also that it only costs 1 comparison to compare b-bit numbers. So we need to be a little careful when we claim that radix sort is linear time.