Today we're going to talk about sorting, which may not come as such a big surprise. We talked about sorting for a while, but we're going to talk about it at a somewhat higher level and question some of the assumptions that we've been making so far. And we're going to ask the question how fast can we sort? A pretty natural question.

You may think you know the answer.

Perhaps you do. Any suggestions on what the answer to this question might be?

There are several possible answers.

Many of them are partially correct.

Let's hear any kinds of answers you'd like and start waking up this fresh morning. Sorry?

Theta n log n. That's a good answer.

That's often correct. Any other suggestions?

N squared. That's correct if all you're allowed to do is swap adjacent elements.

Good. That was close.

I will see if I can make every answer correct.

Usually n squared is not the right answer, but in some models it is. Yeah?

Theta n is also sometimes the right answer.

The real answer is "it depends".

That's the point of today's lecture.

It depends on what we call the computational model, what you're allowed to do. And, in particular here, with sorting, what we care about is the order of the elements, how are you allowed to manipulate the elements, what are you allowed to do with them and find out their order. The model is what you can do with the elements.

Now, we've seen several sorting algorithms.

Do you want to shout some out? I think we've seen four, but maybe you know even more algorithms.

Quicksort. Keep going.

Heapsort. Merge sort.

You can remember all the way back to Lecture 1.

Any others? Insertion sort.

All right. You're on top of it today.

I don't know exactly why, but these two are single words and these two are two words. That's the style.

What is the running time of quicksort?

This is a bit tricky. N log n in the average case.

Or, if we randomize quicksort, randomized quicksort runs in n log n expected for any input sequence.

Let's say n lg n randomized. That's theta.

And the worst-case with plain old quicksort where you just pick the first element as the partition element.

That's n^2. Heapsort, what's the running time there? n lg n always.

Merge sort, I hope you can remember that as well, n lg n. And insertion sort?

n^2. All of these algorithms run no faster than n lg n, so we might ask, can we do better than n lg n?

And that is a question, in some sense, we will answer both yes and no to today.

But all of these algorithms have something in common in terms of the model of what you're allowed to do with the elements. Any guesses on what that model might be? Yeah?

You compare pairs of elements, exactly.

That is indeed the model used by all four of these algorithms.

And in that model n lg n is the best you can do.

We have so far just looked at what are called comparison sorting algorithms or "comparison sorts".

And this is a model for the sorting problem of what you're allowed to do. Here all you can do is use comparisons meaning less than, greater than, less than or equal to, greater than or equal to, equals to determine the relative order of elements.

This is a restriction on algorithms.

It is, in some sense, stating what kinds of elements we're dealing with. They are elements that we can This is a three digit number. somehow compare. They have a total order, some are less, some are bigger.

But is also restricts the algorithm.

You could say, well, I'm sorting integers, but still I'm only allowed to do comparisons with them.

I'm not allowed to multiply the integers or do other weird things. That's the comparison sorting model. And this lecture, in some sense, follows the standard mathematical progression where you have a theorem, Then we get some 4s, then you have a proof, then you have a counter example. It's always a good way to have a math lecture. We're going to prove the theorem that no comparison sorting algorithm runs better than n lg n. Comparisons.

State the theorem, prove that, and then we'll give a counter example in the sense that if you go outside the comparison sorting model you can do better, you can get linear time in some cases, better than n lg n.

So, that is what we're doing today.

But first we're going to stick to this comparison model and try to understand why we need n lg n comparisons if that's all we're allowed to do. And for that we're going to look at something called decision trees, which in some sense is another model of what you're allowed to do in an algorithm, but it's more general than the comparison model. And let's try and example to get some intuition. Suppose we want to sort three elements. This is not very challenging, but we'll get to draw the decision tree that corresponds to sorting three elements. Here is one solution I claim.

This is, in a certain sense, an algorithm, but it's drawn as a tree instead of pseudocode.

What this tree means is that each node you're making a comparison. This says compare a_1 versus a_2. If a_1 is smaller than a_2 you go this way, if it is bigger than a_2 you go this way, and then you proceed. When you get down to a leaf, this is the answer. Remember, the sorting problem is you're trying to find a permutation of the inputs that puts it in sorted order. Let's try it with some sequence of numbers, say 9, 4 and 6.

We want to sort 9, 4 and 6, so first we compare the first element with the second element.

9 is bigger than 4 so we go down this way.

Then we compare the first element with the third element, that's 9 versus 6. 9 is bigger than 6, so we go this way. And then we compare the second element with the third element, 4 is less than 6 and, so we go this way. And the claim is that this is the correct permutation of the elements.

You take a_2, which is 4, then you take a_3, which is 6, and then you take a_1, which is 9, so indeed that works out. And if I wrote this down right, this is a sorting algorithm in the decision tree model.

In general, let me just say the rules of this game.

In general, we have n elements we want to sort.

And I only drew the n = 3 case because these trees get very big very quickly. Each internal node, so every non-leaf node, has a label of the form i : j where i and j are between 1 and n.

And this means that we compare a_i with a_j.

And we have two subtrees from every such node.

We have the left subtree which tells you what the algorithm does, what subsequent comparisons it makes if it comes out less than.

And we have to be a little bit careful because it could also come out equal. What we will do is the left subtree corresponds to less than or equal to and the right subtree corresponds to strictly greater than.

That is a little bit more precise than what we were doing here. Here all the elements were distinct so no problem. But, in general, we care about the equality case too to be general.

So, that was the internal nodes.

And then each leaf node gives you a permutation.

So, in order to be the answer to that sorting problem, that permutation better have the property that it orders the elements. This is from the first lecture when we defined the sorting problem.

Some permutation on n things such that a_pi(1) is less than or equal to a_pi(2) and so on.

So, that is the definition of a decision tree.

Any binary tree with these kinds of labels satisfies all these properties. That is, in some sense, a sorting algorithm. It's a sorting algorithm in the decision tree model. Now, as you might expect, this is really not too different than the comparison model. If I give you a comparison sorting algorithm, we have these four, quicksort, heapsort, merge sort and insertion sort.

All of them can be translated into the decision tree model.

It's sort of a graphical representation of what the algorithm does. It's not a terribly useful one for writing down an algorithm. Any guesses why?

Why do we not draw these pictures as a definition of quicksort or a definition of merge sort?

It depends on the size of the input, that's a good point.

This tree is specific to the value of n, so it is, in some sense, not as generic.

Now, we could try to write down a construction for an arbitrary value of n of one of these decision trees and that would give us sort of a real algorithm that works for any input size.

But even then this is not a terribly convenient representation for writing down an algorithm.

Well, let's write down a transformation that converts a comparison sorting algorithm to a decision tree and then maybe you will see why. This is not a useless model, obviously, I wouldn't be telling you otherwise.

It will be very powerful for proving that we cannot do better than n lg n, but as writing down an algorithm, if you were going to implement something, this tree is not so useful. Even if you had a decision tree computer, whatever that is. But let's prove this theorem that decision trees, in some sense, model comparison sorting algorithms, which we call just comparison sorts.

This is a transformation. And we're going to build one tree for each value of n. The decision trees depend on n.

The algorithm hopefully, well, it depends on n, but it works for all values of n.

And we're just going to think of the algorithm as splitting into two forks, the left subtree and the right subtree whenever it makes a comparison.

If we take a comparison sort like merge sort.

And it does lots of stuff. It does index arithmetic, it does recursion, whatever.

But at some point it makes a comparison and then we say, OK, there are two halves of the algorithm.

There is what the algorithm would do if the comparison came out less than or equal to and what the algorithm would do if the comparison came out greater than.

So, you can build a tree in this way.

In some sense, what this tree is doing is listing all possible executions of this algorithm considering what would happen for all possible values of those comparisons.

We will call these all possible instruction traces.

If you write down all the instructions that are executed by this algorithm, for all possible input arrays, a_1 to a_n, see what all the comparisons, how they could come and what the algorithm does, in the end you will get a tree.

Now, how big will that tree be roughly?

As a function of n. Yeah?

Right. If it's got to be able to sort every possible list of length n, at the leaves I have to have all the permutations of those elements.

That is a lot. There are a lot of permeations on n elements. There's n factorial of them.

N factorial is exponential, it's really big.

So, this tree is huge. It's going to be exponential on the input size n. That is why we don't write algorithms down normally as a decision tree, even though in some cases maybe we could.

It's not a very compact representation.

These algorithms, you write them down in pseudocode, they have constant length.

It's a very succinct representation of this algorithm. Here the length depends on n and it depends exponentially on n, which is not useful if you wanted to implement the algorithm because writing down the algorithm would take a long time.

But, nonetheless, we can use this as a tool to analyze these comparison sorting algorithms.

We have all of these. Any algorithm can be transformed in this way into a decision tree.

And now we have this observation that the number of leaves in this decision tree has to be really big.

Let me talk about leaves in a second.

Before we get to leaves, let's talk about the depth of the tree.

This decision tree represents all possible executions of the algorithm. If I look at a particular execution, which corresponds to some root to leaf path in the tree, the running time or the number of comparisons made by that execution is just the length of the path.

And, therefore, the worst-case running time, over all possible inputs of length n, is going to be -- n - 1? Could be.

Depends on the decision tree. But, as a function of the decision tree? The longest path, right, which is called the height of the tree.

So, this is what we want to measure.

We want to claim that the height of the tree has to be at least n lg n with an omega in front.

That is what we'll prove.

And the only thing we're going to use is that the number of leaves in that tree has to be big, has to be n factorial.

This is a lower bound on decision tree sorting.

And the lower bound says that if you have any decision tree that sorts n elements then its height has to be at least n lg n up to constant factors.

So, that is the theorem. Now we're going to prove the theorem. And we're going to use that the number of leaves in that tree must be at least n factorial.

Because there are n factorial permutations of the inputs.

All of them could happen. And so, for this algorithm to be correct, it has detect every one of those permutations in some way. Now, it may do it very quickly.

We better only need n lg n comparisons because we know that's possible. The depth of the tree may not be too big, but it has to have a huge number of leaves down there. It has to branch enough to get n factorial leaves because it has to give the right answer in possible inputs. This is, in some sense, counting the number of possible inputs that we have to distinguish. This is the number of leaves.

What we care about is the height of the tree.

Let's call the height of the tree h.

Now, if I have a tree of height h, how many leaves could it have? What's the maximum number of leaves it could have?

2^h, exactly. Because this is binary tree, comparison trees always have a branching factor of 2, the number of leaves has to be at most 2^h, if I have a height h tree. Now, this gives me a relation.

The number of leaves has to be greater than or equal to n factorial and the number of leaves has to be less than or equal to 2^h. Therefore, n factorial is less than or equal to 2^h, if I got that right.

Now, again, we care about h in terms of n factorial, so we solve this by taking logs.

And I am also going to flip sides.

Now h is at least log base 2, because there is a 2 over here, of n factorial. There is a property that I'm using here in order to derive this inequality from this inequality. This is a technical aside, but it's important that you realize there is a technical issue here.

The general principle I'm applying is I have some inequality, I do the same thing to both sides, and hopefully that inequality should still be true.

But, in order for that to be the case, I need a property about that operation that I'm performing.

It has to be a monotonic transformation.

Here what I'm using is that log is a monotonically increasing function. That is important.

If I multiply both sides by -1, which is a decreasing function, the inequality would have to get flipped.

The fact that the inequality is not flipping here, I need to know that log is monotonically increasing.

If you see log that's true. We need to be careful here.

Now we need some approximation of n factorial in order to figure out what its log is. Does anyone know a good approximation for n factorial? Not necessarily the equation but the name. Stirling's formula.

Good. You all remember Stirling.

And I just need the highest order term, which I believe is that. N factorial is at least (n/e)^n. So, that's all we need here.

Now I can use properties of logs to bring the n outside.

This is n lg (n/e).

And then lg (n/e) I can simplify.

That is just lg n - lg e. So, this is n(lg n - lg e).

Lg e is a constant, so it's really tiny compared to this lg n which is growing within.

This is Omega(n lg n). All we care about is the leading term. It is actually Theta(n lg n), but because we have it greater than or equal to all we care about is the omega. A theta here wouldn't give us anything stronger. Of course, not all algorithms have n lg n running time or make n lg n comparisons.

Some of them do, some of them are worse, but this proves that all of them require a height of at least n lg n. There you see proof, once you observe the fact about the number of leaves, and if you remember Stirling's formula.

So, you should know this proof. You can show that all sorts of problems require n lg n time with this kind of technique, provided you're in some kind of a decision tree model.

That's important. We really need that our algorithm can be phrased as a decision tree.

And, in particular, we know from this transformation that all comparison sorts can be represented as the decision tree.

But there are some sorting algorithms which cannot be represented as a decision tree. And we will turn to that momentarily. But before we get there I phrased this theorem as a lower bound on decision tree sorting.

But, of course, we also get a lower bound on comparison sorting. And, in particular, it tells us that merge sort and heapsort are asymptotically optimal. Their dependence on n, in terms of asymptotic notation, so ignoring constant factors, these algorithms are optimal in terms of growth of n, but this is only in the comparison model.

So, among comparison sorting algorithms, which these are, they are asymptotically optimal.

They use the minimum number of comparisons up to constant factors. In fact, their whole running time is dominated by the number of comparisons.

It's all Theta(n lg n). So, this is good news.

And I should probably mention a little bit about what happens with randomized algorithms. What I've described here really only applies, in some sense, to deterministic algorithms. Does anyone see what would change with randomized algorithms or where I've assumed that I've had a deterministic comparison sort?

This is a bit subtle. And I only noticed it reading the notes this morning, oh, wait.

I will give you a hint. It's over here, the right-hand side of the world.

If I have a deterministic algorithm, what the algorithm does is completely determinate at each step.

As long as I know all the comparisons that it made up to some point, it's determinate what that algorithm will do.

But, if I have a randomized algorithm, it also depends on the outcomes of some coin flips. Any suggestions of what breaks over here? There is more than one tree, exactly. So, we had this assumption that we only have one tree for each n.

In fact, what we get is a probability distribution over trees. For each value of n, if you take all the possible executions of that algorithm, all the instruction traces, well, now, in addition to branching on comparisons, we also branch on whether a coin flip came out heads or tails, or however we're generating random numbers it came out with some value between 1 and n. So, we get a probability distribution over trees. This lower bound still applies, though. Because, no matter what tree we get, I don't really care. I get at least one tree for each n. And this proof applies to every tree. So, no matter what tree you get, if it is a correct tree it has to have height Omega(n lg n). This lower bound applies even for randomized algorithms. You cannot get better than n lg n, because no matter what tree it comes up with, no matter how those coin flips come out, this argument still applies. Every tree that comes out has to be correct, so this is really at least one tree.

And that will now work. We also get the fact that randomized quicksort is asymptotically optimal in expectation.

But, in order to say that randomized quicksort is asymptotically optimal, we need to know that all randomized algorithms require Omega(n lg n) comparisons.

Now we know that so all is well.

That is the comparison model. Any questions before we go on?

Good. The next topic is to burst outside of the comparison model and try to sort in linear time.

It is pretty clear that, as long as you don't have some kind of a parallel algorithm or something really fancy, you cannot sort any better than linear time because you've at least got to look at the data. No matter what you're doing with the data, you've got to look at it, otherwise you're not sorting it correctly.

So, linear time is the best we could hope for.

N lg n is pretty close. How could we sort in linear time? Well, we're going to need some more powerful assumption. And this is the counter example. We're going to have to move outside the comparison model and do something else with our elements. And what we're going to do is assume that they're integers in a particular range, and we will use that to sort in linear time.

We're going to see two algorithms for sorting faster than n lg n. The first one is pretty simple, and we will use it in the second algorithm.

It's called counting sort. The input to counting sort is an array, as usual, but we're going to assume what those array elements look like. Each A[i] is an integer from the range of 1 to k. This is a pretty strong assumption. And the running time is actually going to depend on k. If k is small it is going to be a good algorithm. If k is big it's going to be a really bad algorithm, worse than n lg n.

Our goal is to output some sorted version of this array.

Let's call this sorting of A. It's going to be easier to write down the output directly instead of writing down permutation for this algorithm. And then we have some auxiliary storage. I'm about to write down the pseudocode, which is why I'm declaring all my variables here.

And the auxiliary storage will have length k, which is the range on my input values.

Let's see the algorithm.

This is counting sort.

And it takes a little while to write down but it's pretty straightforward.

First we do some initialization.

Then we do some counting.

Then we do some summing.

And then we actually write the output.

Is that algorithm perfectly clear to everyone?

No one. Good. This should illustrate how obscure pseudocode can be.

And when you're solving your problem sets, you should keep in mind that it's really hard to understand an algorithm just given pseudocode like this.

You need some kind of English description of what's going on because, while you could work through and figure out what this means, it could take half an hour to an hour.

And that's not a good way of expressing yourself.

And so what I will give you now is the English description, but we will refer back to this to understand.

This is sort of our bible of what the algorithm is supposed to do. Let me go over it briefly.

The first step is just some initialization.

The C[i]'s are going to count some things, count occurrences of values. And so first we set them to zero. Then, for every value we see A[j], we're going to increment the counter for that value A[j].

Then the C[i]s will give me the number of elements equal to a particular value i. Then I'm going to take prefix sums, which will make it so that C[i] gives me the number of keys, the number of elements less than or equal to [i] instead of equals. And then, finally, it turns out that's enough to put all the elements in the right place. This I will call distribution.

This is the distribution step. And it's probably the least obvious of all the steps. And let's do an example to make it more obvious what's going on.

Let's take an array A = [4, 1, 3, 4, 3].

And then I want some array C. And let me add some indices here so we can see what the algorithm is really doing.

Here it turns out that all of my numbers are in the range 1 to 4, so k = 4. My array C has four values.

Initially, I set them all to zero.

That's easy. And now I want to count through everything. And let me not cheat here.

I'm in the second step, so to speak.

And I look for each element in order.

I look at the C[i] value. The first element is 4, so I look at C4. That is 0.

I increment it to 1. Then I look at element 1.

That's 0. I increment it to 1.

Then I look at 3 and that's here.

It is also 0. I increment it to 1.

Not so exciting so far. Now I see 4, which I've seen before, how exciting.

I had value 1 in here, I increment it to 2.

Then I see value 3, which also had a value of 1.

I increment that to 2. The result is [1, 0, 2, 2]. That's what my array C looks like at this point in the algorithm.

Now I do a relatively simple transformation of taking prefix sums. I want to know, instead of these individual values, the sum of this prefix, the sum of this prefix, the sum of this prefix and the sum of this prefix. I will call that C prime just so we don't get too lost in all these different versions of C.

This is just 1. And 1 plus 0 is 1.

1 plus 2 is 3. 3 plus 2 is 5.

So, these are sort of the running totals.

There are five elements total, there are three elements less than or equal to 3, there is one element less than or equal to 2, and so on.

Now, the fun part, the distribution.

And this is where we get our array B.

B better have the same size, every element better appear here somewhere and they should come out in sorted order.

Let's just run the algorithm. j is going to start at the end of the array and work its way down to 1, the beginning of the array. And what we do is we pick up the last element of A, A[n].

We look at the counter. We look at the C vector for that value. Here the value is 3, and this is the third column, so that has number 3.

And the claim is that's where it belongs in B.

You take this number 3, you put it in index 3 of the array B. And then you decrement the counter. I'm going to replace 3 here with 2. And the idea is these numbers tell you where those values should go.

Anything of value 1 should go at position 1.

Anything with value 3 should go at position 3 or less.

This is going to be the last place that a 3 should go.

And then anything with value 4 should go at position 5 or less, definitely should go at the end of the array because 4 is the largest value. And this counter will work out perfectly because these counts have left enough space in each section of the array. Effectively, this part is reserved for ones, there are no twos, this part is reserved for threes, and this part is reserved for fours. You can check if that's really what this array means. Let's finish running the algorithm. That was the last element.

I won't cross it off, but we've sort of done that.

Now I look at the next to last element.

That's a 4. Fours go in position 5.

So, I put my 4 here in position 5 and I decrement that counter.

Next I look at another 3. Threes now go in position 2, so that goes there. And then I decrement that counter. I won't actually use that counter anymore, but let's decrement it because that's what the algorithm says. I look at the previous element.

That's a 1. Ones go in position 1, so I put it here and decrement that counter.

And finally I have another 4. And fours go in position 4 now, position 4 is here, and I decrement that counter.

So, that's counting sort. And you'll notice that all the elements appear and they appear in order, so that's the algorithm. Now, what's the running time of counting sort? kn is an upper bound.

It's a little bit better than that.

Actually, quite a bit better. This requires some summing.

Let's go back to the top of the algorithm.

How much time does this step take?

k. How much time does this step take? n.

How much time does this step take?

k. Each of these operations in the for loops is taking constant time, so it is how many iterations of that for loop are there?

And, finally, this step takes n.

So, the total running time of counting sort is k + n.

And this is a great algorithm if k is relatively small, like at most n. If k is big like n^2 or 2^n or whatever, this is not such a good algorithm, but if k = O(n) this is great. And we get our linear time sorting algorithm. Not only do we need the assumption that our numbers are integers, but we need that the range of the integers is pretty small for this algorithm to work. If all the numbers are between 1 and order n then we get a linear time algorithm.

But as soon as they're up to n lg n we're toast.

We're back to n lg n sorting. It's not so great.

So, you could write a combination algorithm that says, well, if k is bigger than n lg n, then I will just use merge sort. And if it's less than n lg n I'll use counting sort. And that would work, but we can do better than that. How's the time?

It is worth noting that we've beaten our bound, but only assuming that we're outside the comparison model.

We haven't really contradicted the original theorem, we're just changing the model. And it's always good to question what you're allowed to do in any problem scenario.

In, say, some practical scenarios, this would be great if the numbers you're dealing with are, say, a byte long. Then k is only 2^8, which is 256. You need this auxiliary array of size 256, and this is really fast.

256 + n, no matter how big n is it's linear in n.

If you know your numbers are small, it's great.

But if you're numbers are bigger, say you still know they're integers but they fit in like 32 bit words, then life is not so easy. Because k is then 2^32, which is 4.2 billion or so, which is pretty big.

And you would need this auxiliary array of 4.2 billion words, which is probably like 16 gigabytes.

So, you just need to initialize that array before you can even get started. Unless n is like much, much more than 4 billion and you have 16 gigabytes of storage just to throw away, which I don't even have any machines with 16 gigabytes of RAM, this is not such a great algorithm. Just to get a feel, it's good, the numbers are really small.

What we're going to do next is come up with a fancier algorithm that uses this as a subroutine on small numbers and combines this algorithm to handle larger numbers.

That algorithm is called radix sort.

But we need one important property of counting sort before we can go there.

And that important property is stability.

A stable sorting algorithm preserves the order of equal elements, let's say the relative order.

This is a bit subtle because usually we think of elements just as numbers. And, yeah, we had a couple threes and we had a couple fours.

It turns out, if you look at the order of those threes and the order of those fours, we kept them in order. Because we took the last three and we put it here. Then we took the next to the last three and we put it to the left of that where O is decrementing our counter and moving from the end of the array to the beginning of the array. No matter how we do that, the orders of those threes are preserved, the orders of the fours are preserved. This may seem like a relatively simple thing, but if you look at the other four sorting algorithms we've seen, not all of them are stable. So, this is an exercise.

Exercise is figure out which other sorting algorithms that we've seen are stable and which are not.

I encourage you to work that out because this is the sort of thing that we ask on quizzes. But for now all we need is that counting sort is stable. And I won't prove this, but it should be pretty obvious from the algorithm.

Now we get to talk about radix sort.

Radix sort is going to work for a much larger range of numbers in linear time. Still it has to have an assumption about how big those numbers are, but it will be a much more lax assumption. Now, to increase suspense even further, I am going to tell you some history about radix sort.

This is one of the oldest sorting algorithms.

It's probably the oldest implemented sorting algorithm.

It was implemented around 1890. This is Herman Hollerith.

Let's say around 1890. Has anyone heard of Hollerith before? A couple people.

Not too many. He is sort of an important guy.

He was a lecturer at MIT at some point.

He developed an early version of punch cards.

Punch card technology. This is before my time so I even have to look at my notes to remember.

Oh, yeah, they're called punch cards.

You may have seen them. If not they're in the PowerPoint lecture notes. There's this big grid.

These days, if you've used a modern punch card recently, they are 80 characters wide and, I don't know, I think it's something like 16, I don't remember exactly.

And then you punch little holes here.

You have this magic machine. It's like a typewriter.

You press a letter and that corresponds to some character.

Maybe it will punch out a hole here, punch out a hole here.

You can see the website if you want to know exactly how this works for historical reasons. You don't see these too often anymore, but this is in particular the reason why most terminals are 80 characters wide because that was how things were. Hollerith actually didn't develop these punch cards exactly, although eventually he did. In the beginning, in 1890, the big deal was the US Census.

If you watched the news, I guess like a year or two ago, the US Census was a big deal because it's really expensive to collect all this data from everyone.

And the Constitution says you've got to collect data about everyone every ten years. And it was getting hard.

In particular, in 1880, they did the census.

And it took them almost ten years to complete the census.

The population kept going up, and ten years to do a ten-year census, that's going to start getting expensive when they overlap with each other. So, for 1890 they wanted to do something fancier. And Hollerith said, OK, I'm going to build a machine that you take in the data. It was a modified punch card where you would mark out particular squares depending on your status, whether you were single or married or whatever.

All the things they wanted to know on the census they would encode in binary onto this card. And then he built a machine that would sort these cards so you could do counting.

And, in some sense, these are numbers.

And the numbers aren't too big, but they're big enough that counting sort wouldn't work. I mean if there were a hundred numbers here, 2^100 is pretty overwhelming, so we cannot use counting sort. The first idea was the wrong idea. I'm going to think of these as numbers. Let's say each of these columns is one number. And so there's sort of the most significant number out here and there is the least significant number out here. The first idea was you sort by the most significant digit first.

That's not such a great algorithm, because if you sort by the most significant digit you get a bunch of buckets each with a pile of cards. And this was a physical device.

It wasn't exactly an electronically controlled computer. It was a human that would push down some kind of reader. It would see which holes in the first column are punched. And then it would open a physical bin in which the person would sort of swipe it and it would just fall into the right bin.

It was a semi-automated. I mean the computer was the human plus the machine, but never mind.

This was the procedure. You sorted it into bins.

Then you had to go through and sort each bin by the second digit. And pretty soon the number of bins gets pretty big. And if you don't have too many digits this is OK, but it's not the right thing to do. The right idea, which is what Hollerith came up with after that, was to sort by the least significant digit first.

And you should also do that using a stable sorting algorithm. Now, Hollerith probably didn't call it a stable sorting algorithm at the time, but we will. And this won Hollerith lots of money and good things. He founded this tabulating machine company in 1911, and that merged with several other companies to form something you may have heard of called IBM in 1924. That may be the context in which you've heard of Hollerith, or if you've done punch cards before. The whole idea is that we're doing a digit by digit sort. I should have mentioned that at the beginning. And we're going to do it from least significant to most significant.

It turns out that works. And to see that let's do an example. I think I'm going to need a whole two boards ideally.

First we'll see an example.

Then we'll prove the theorem. The proof is actually pretty darn easy. But, nonetheless, it's rather counterintuitive this works if you haven't seen it before. Certainly, the first time I saw it, it was quite a surprise. The nice thing also about this algorithm is there are no bins. It's all one big bin at all times. Let's take some numbers.

I'm spacing out the digits so we can see them a little bit better. 657, 839, 436, 720 and 355.

I'm assuming here we're using decimal numbers.

Why not? Hopefully this are not yet sorted. We'd like to sort them.

The first thing we do is take the least significant digit, sort by the least significant digit.

And whenever we have equal elements like these two nines, we preserve their relative order.

So, 329 is going to remain above 839.

It doesn't matter here because we're doing the first sort, but in general we're always using a stable sorting algorithm. When we sort by this column, first we get the zero, so that's 720, then we get 5, Then we get 6, Stop me if I make a mistake. Then we get the 7s, and we preserve the order. Here it happens to be the right order, but it may not be at this point.

We haven't even looked at the other digits.

Then we get 9s, there are two 9s, 329 and 839. All right so far?

Good. Now we sort by the middle digit, the next least significant.

And we start out with what looks like the 2s.

There is a 2 up here and a 2 down here.

Of course, we write the first 2 first, 720, then 329.

Then we have the 3s, so we have 436 and 839.

Then we have a bunch of 5s it looks like.

Have I missed anyone so far? No.

Good. We have three 5s, 355, 457 and 657. I like to check that I haven't lost any elements. We have seven here, seven here and seven elements here.

Good. Finally, we sort by the last digit. One thing to notice, by the way, is before we sorted by the last digit -- Currently these numbers don't resemble sorted order at all.

But if you look at everything beyond the digit we haven't yet sorted, so these two digits, that's nice and sorted, 20, 29, 36, 39, 55, 57, 57.

Pretty cool. Let's finish it off.

We stably sort by the first digit.

And the smallest number we get is a 3, so we get 329 and then 436 and 457, then we get a 6, 657, then a 7, and then we have an 8.

And check. I still have seven elements.

Good. I haven't lost anyone.

And, indeed, they're now in sorted order.

And you can start to see why this is working.

When I have equal elements here, I have already sorted the suffix. Let's write down a proof of that. What is nice about this algorithm is we're not partitioning into bins.

We always keep the huge batch of elements in one big pile, but we're just going through it multiple times.

In general, we sort of need to go through it multiple times.

Hopefully not too many times. But let's first argue correctness. To analyze the running time is a little bit tricky here because it depends how you partition into digits. Correctness is easy.

We just induct on the digit position that we're currently sorting, so let's call that t. And we can assume by induction that it's sorted beyond digit t. This is our induction hypothesis. We assume that we're sorted on the low-order t - 1 digits. And then the next thing we do is sort on the t-th digit. We just need to check that things work. And we restore the induction hypothesis for t instead of t - When we sort on the t-th digit there are two cases.

If we look at any two elements, we want to know whether they're put in the right order. If two elements are the same, let's say they have the same t-th digit -- This is the tricky case. If they have the same t-th digit then their order should not be changed.

So, by stability, we know that they remain in the same order because stability is supposed to preserve things that have the same key that we're sorting on.

And then, by the induction hypothesis, we know that that keeps them in sorted order because induction hypothesis says that they used to be sorted.

Adding on this value in the front that's the same in both doesn't change anything so they remain sorted.

And if they have differing t-th digits -- -- then this sorting step will put them in the right order.

Because that's what sorting does.

This is the most significant digit, so you've got to order them by the t-th digit if they differ.

The rest are irrelevant. So, proof here of correctness is very simple once you know the algorithm.

Any questions before we go on? Good.

We're going to use counting sort.

We could use any sorting algorithm we want for individual digits, but the only algorithm that we know that runs in less than n lg n time is counting sort.

So, we better use that one to sort of bootstrap and get an even faster and more general algorithm.

I just erased the running time. Counting sort runs in order k + n time. We need to remember that.

And the range of the numbers is 1 to k or 0 to k - 1.

When we sort by a particular digit, we shouldn't use n lg n algorithm because then this thing will take n lg n for one round and it's going to have multiple rounds.

That's going to be worse than n lg n.

We're going to use counting sort for each round.

We use counting sort for each digit.

And we know the running time of counting sort here is order k + n . But I don't want to assume that my integers are split into digits for me.

That's sort of giving away too much flexibility.

Because if I have some number written in whatever form it is, probably written in binary, I can cluster together some of those bits and call that a digit.

Let's think of our numbers as binary.

Suppose we have n integers. And they're in some range.

And we want to know how big a range they can be.

Let's say, a sort of practical way of thinking, you know, we're in a binary world, each integer is b bits long. So, in other words, the range is from 0 to 2b - 1. I will assume that my numbers are non-negative. It doesn't make much difference if they're negative, too.

I want to know how big a b I can handle, but I don't want to split into bits as my digits because then I would have b digits and I would have to do b rounds of this algorithm.

The number of rounds of this algorithm is the number of digits that I have. And each one costs me, let's hope, for linear time. And, indeed, if I use a single bit, k = 2.

And so this is order n. But then the running time would be order n per round. And there are b digits, if I consider them to be bits, order n times b time.

And even if b is something small like log n, if I have log n bits, then these are numbers between 0 and n - 1. I already know how to sort numbers between 0 and n - 1 in linear time.

Here I'm spending n lg n time, so that's no good.

Instead, what we're going to do is take a bunch of bits and call that a digit, the most bits we can handle with counting sort. The notation will be I split each integer into b/r digits. Each r bits long.

In other words, I think of my number as being in base 2^r. And I happen to be writing it down in binary, but I cluster together r bits and I get a bunch of digits in base 2^r.

And then there are b/ r digits. This b/r is the number of rounds. And this base -- This is the maximum value I have in one of these digits.

It's between 0 and 2^r. This is, in some sense, k for a run of counting sort.

What is the running time? Well, I have b/r rounds.

It's b/r times the running time for a round.

Which I have n numbers and my value of k is 2^r.

This is the running time of counting sort, n + k, this is the number of rounds, so this is b/r (n+2^r).

And I am free to choose r however I want.

What I would like to do is minimize this run time over my choices of r. Any suggestions on how I might find the minimum running time over all choices of r?

Techniques, not necessarily solutions.

We're not used to this because it's asymptomatic, but forget the big O here. How do I minimize a function with respect to one variable? Take the derivative, yeah. I can take the derivative of this function by r, differentiate by r, set the derivative equal to 0, and that should be a critical point in this function. It turns out this function is unimodal in r and you will find the minimum.

We could do that. I'm not going to do it because it takes a little bit more work. You should try it at home.

It will give you the exact minimum, which is good if you know what this constant is. Differentiate with respect to r and set to 0. I am going to do it a little bit more intuitively, in other words less precisely, but I will still get the right answer.

And definitely I will get an upper bound because I can choose r to be whatever I want. It turns out this will be the right answer. Let's just think about growth in terms of r. There are essentially two terms here. I have b/r(n) and I have b/r(2^r). Now, b/r(n) would like r to be as big as possible. The bigger r is the number of rounds goes down. This number in front of n, this coefficient in front of n goes down, so I would like r to be big. So, b/r(n) wants r big.

However, r cannot be too big. This is saying I want digits that have a lot of bits in them. It cannot be too big because there's 2^r term out here. If this happens to be bigger than n then this will dominate in terms of growth of r.

This is going to be b times 2 to the r over r.

2 the r is much, much bigger than r, so it's going to grow much faster is what I mean.

And so I really don't want r to be too big for this other term.

So, that is b/4(2^r) wants r small.

Provided that this term is bigger or equal to this term then I can set r pretty big for that term.

What I want is the n to dominate the 2^r.

Provided I have that then I can set r as large as I want.

Let's say I want to choose r to be maximum subject to this condition that n is greater than or equal to 2^r.

This is an upper bound to 2^r, and upper bound on r.

In other words, I want r = lg n.

This turns out to be the right answer up to constant factors.

There we go. And definitely choosing r to be lg n will give me an upper bound on the best running time I could get because I can choose it to be whatever I want.

If you differentiate you will indeed get the same answer.

This was not quite a formal argument but close, because the big O is all about what grows fastest.

If we plug in r = lg n we get bn/lg n.

The n and the 2^r are equal, that's a factor of 2, 2 times n, not a big deal. It comes out into the O.

We have bn/lg n which is r. We have to think about what this means and translate it in terms of range.

b was the number of bits in our number, which corresponds to the range of the number. I've got 20 minutes under so far in lecture so I can go 20 minutes over, right? No, I'm kidding.

Almost done. Let's say that our numbers, are integers are in the range, we have 0 to 2^b, I'm going to say that it's range 0 to nd.

This should be a -1 here. If I have numbers that are between 0 and n^d - 1 where d is a constant or d is some parameter, so this is a polynomial in n, then you work out this running time.

It is order dn. This is the way to think about it because now we can compare to counting sort.

Counting sort could handle 0 up to some constant times d in linear time. Now I can handle 0 up to n to some constant power in linear time.

This is if d = order 1 then we get a linear time sorting algorithm. And that is cool as long as d is at most lg n. As long as your numbers are at most n lg n then we have something that beats our n lg n sorting algorithms. And this is pretty nice.

Whenever you know that your numbers are order log end bits long we are happy, and you get some smooth tradeoff there. For example, if we have our 32 bit numbers and we split into let's say eight bit chunks then we'll only have to do four rounds each linear time and we have just 256 working space.

We were doing four rounds for 32 bit numbers.

If you use n lg n algorithm, you're going to be doing lg n rounds through your numbers. n is like 2000, and that's at least 11 rounds for example.

You would think this algorithm is going to be much faster for small numbers. Unfortunately, counting sort is not very good on a cache.

In practice, rating sort is not that fast an algorithm unless your numbers are really small.

Something like quicksort can do better.

It's sort of shame, but theoretically this is very beautiful. And there are contexts where this is really the right way to sort things.

I will mention finally that if you have arbitrary integers that are one word length long. Here we're assuming that there are b bits in a word and we have some depends indirectly on b here. But, in general, if you have a bunch of integers and they're one word length long, and you can manipulate a word in constant time, then the best algorithm we know for sorting runs in n times square root of lg lg n time expected.

It is a randomized algorithm. We're not going to cover that algorithm in this class. It's rather complicated.

I didn't even cover it in Advanced Algorithms when I taught it. If you want something easier, you can get n times square root of lg lg n time worst-case.

And that paper is almost readable.

I have taught that in Advanced Algorithms.

If you're interested in this kind of stuff, take Advanced Algorithms next fall.

It's one of the follow-ons to this class.

These are much more complicated algorithms, but it gives you some sense. You can even break out of the dependence on b, as long as you know that b is at most a word. And I will stop there unless there are any questions. Then see you Wednesday.