Integer sorting

Last lecture we saw a general lower bound which says that any *comparison model* sorting algorithm costs at least $\Omega(n \log n)$ in the worst case. In this lecture we will derive sorting algorithms that run in $O(n)$ cost! The catch is that to beat the comparison model lower bound we must of course leave the comparison model behind and explore other models of computation. Specifically, we will be looking at the problem of sorting integers, which gives us more power then the comparison model since we can use properties of integers to help us sort them faster. We will show two algorithms, Counting Sort and Radix Sort, which are capable of sorting integers in linear time, provided that those integers are not too large.

Objectives of this lecture

In this lecture, we want to:

- See the *Word RAM model of computation* for integer (non-comparison) algorithms
- Learn about the *counting sort* algorithm for sorting (small) integers
- Learn about the *radix sort* algorithm for sorting (slightly bigger) integers

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 8: Sorting in Linear Time

1 Models of computation for integers

In the first two lectures we have predominantly concerned ourselves with algorithms in the *comparison model* of computation, where the input to our algorithms consisted of an array of n comparable elements. In this model we didn't have any other information available to us about the elements. Were they integers, numbers, strings, who knows? This made the model very general since we could derive algorithms for sorting and selection that effectively work on *any type* because absolutely no assumptions about the type were made, only that they were comparable, which is essentially required by the definition of sorting!

Last lecture we saw a very cool and fundamental result which is that in the comparison model, sorting *can not* be done in less than $\Omega(n \log n)$ cost. We proved that it was mathematically impossible to invent a sorting algorithm that runs faster than this. Today, we want to invent some sorting algorithms than run in $O(n)$ cost! To do so without contradicting ourselves we are going to have to leave the comparison model behind and explore models of computation that give us more power. Specifically, we are going to work on the problem of sorting *integers*, which means that unlike in the comparison model where all we could do was compare two elements, we will now gain the power to do things to the input elements like add them, divide them, use them as indices into arrays, etc. These new found powers, and in particular the last one (using the items as indices) will be the tools to beating the comparison model. Our model of computation for today (and implicitly for much of the rest of the course) is the *word RAM model*.

1.1 The Word RAM model of computation

Definition: Word RAM model

In the word RAM model:

- We have unlimited constant-time addressable memory (called "registers"),
- Each register can store a w-bit integer (called a "word"),
- Reading/writing, arithmetic, logic, bitwise operations on a constant number of words takes constant time,
- With input size n, we need $w \geq \log n$.

The final assumption is needed because if our input contains n words, then to be capable of even reading the contents of the input, we are surely going to need to be able to write down the integer n as an index, and that requires $\log n$ bits. Since the word size is w, the maximum integer we can store in a single word $2^w - 1$.

Note that unlike some of our previous models such as the comparison model which had concrete costs, e.g., exactly 1 per comparison, in the word RAM we only care about asymptotic costs, so we ignore constant factors and just say that operations on a constant number of words takes constant time. This model is essentially just a more formal version of what you are probably used to from your previous classes when you analyzed an algorithm by counting "instruc-

tions". The only subtle part is the restriction on the word size w and the assumption that only operations on w-bit integers take constant time. Most of the time this is of no consequence, but there are some situations where it matters.

The importance of the word size Consider for example an algorithm that takes n integers as inputs each of which is written with w bits, and computes their product. How does such an algorithm work and what is its runtime? A "count the instructions" analysis would suggest that it just multiplies all the numbers together and takes $O(n)$ time!. However, remember that for n integers each w bits, their product is an integer containing nw bits, which requires n registers to store! Computing this product would therefore take much more than $\Theta(n)$ time since multiplying a super-constant number of integers can not be done in a single instruction and would instead require an algorithm for multiplying large integers[1].

This might seem odd at first but this model is really trying to help us match the behavior that such an algorithm would have on a real computer! Almost every modern processor operates on 64-bit words, such that all arithmetic, bitwise, comaprison operations, etc., can be done with a single machine instruction. What would happen if you tried to implement an algorithm that multiplies n integers on a real computer? In most languages other than Python, you will quickly find that the result will *overflow*, so you will just get a wrong answer (most likely you'll get the answer modulo 2^{64} or something similar.) In Python you will find that you do in fact get the right answer, but the computation will become very slow! Under the hood Python is actually happy to represent large numbers for you by decomposing them into an array of word-sized integers. Doing arithmetic on these big integers takes more than constant time. Python uses an algorithm called *Karatsuba multiplcation* for multiplying these big integers, which runs in $O(d^{\log_2 3}) \approx O(d^{1.58})$ operations for d-digit numbers.

But what if... we ignore the word size An alternative model is the *unit-cost RAM model* which does not place any restriction on the size of the integers. In this model we just say that all arithmetic operations on integers takes constant time regardless of how many bits it takes to represent them. This might seem like an unimportant and pedantic difference, but it turns out that this assumption allows you to implement some wild and crazy algorithms, such as being able to sort n integers in constant time![2] Perhaps even more ludicrous, a RAM with unlimited precision (no bound on w) can solve PSPACE-Complete problems in polynomial time![3]

1.2 Beating the comparison model

To beat the comparison model, we have to advantage of some power that it doesn't have. The major limitation of the comparison model is that every operation we pay for (a comparison) can only result in a *binary outcome*. This is fundamentally why our information theoretic lower bounds gave us $\log_2 \#$ outputs, because the best we could do was double/halve the possible outcomes each time we paid a cost of one. The source of untapped power of the word RAM is that we can use our input elements (integers) as indices into arrays! That is, if I have an array

[1] We might see an algorithm for this later in the course. It takes $\Theta(d \log d)$ instructions to multiply d-digit integers!
[2] Appendix A of Computing with arbitrary and random numbers, Michael Brand's PhD thesis, Monash University.
[3] See *A characterization of the class of functions computable in polynomial time on Random Access Machines*

of length n and some integer from 1 to n (or 0 to $n-1$ if zero-indexed), then I can access the value of the array in constant time, but depending on the value of the integer, there are now n possible outcomes, which is far more than the two outcomes of the comparison model!

Warmup example: constant-time static search Before we dive into sorting, lets demonstrate how the word RAM has more power than the comparison model. The same ideas will be used momentarily in our sorting algorithms. Consider the simpler problem of *static searching*, i.e., outputting the position of a given element in a given array if it exists, where arbitrary preprocessing is allowed. By arbitrary preprocessing, we mean that you can, for example, sort the elements or arrange them into a binary search tree, all for free to make the queries faster to answer. There are $n+1$ possible outcomes (each position and "it doesn't exist"), so an information theoretic lower bound in the comparison model immediately tells us that we can't do better than $\Omega(\log n)$ cost. This tells us that binary search (on a sorted array) and balanced binary search trees are *asymptotically optimal* in the comparison model as they match this bound.

But what could we do with the power of the word RAM when our elements are word-size integers? Given an array $a_1, \ldots, a_n$ of n integers which are in the range $\{0, 1, \ldots, u-1\}$, where $u \leq 2^w$ is the size of the *universe* of inputs we are considering, we could create an array T of size u, one slot for every possible value, then store $T[a_i] \leftarrow i$, i.e., store a lookup table of positions based on the values. To answer a search query for a value x we simply check $T[x]$ and output the index if there is one stored there. This solves the problem of static searching in constant time! We have defeated the confines of the comparison model, at the expense of making the assumption that our keys are integers rather than arbitrary comparable things. Furthermore this solution makes the assumption that u is a reasonable amount of space to use. If the universe of keys is very large, this solution is horribly space inefficient. That problem can be effectively eliminated by the use of *hashing* which we will study in great detail in the next few lectures!

2 Sorting small integers: Counting Sort

Lets start by setting up the problem precisely.

Problem: Integer sorting

The integer sorting problem consists of an input array of n elements $a_1, a_2, \ldots, a_n$, each identified by a (not necessarily unique) integer key called key(a_i). The goal is to output an array containing a permutation of the input $a_{\pi_1}, \ldots, a_{\pi_n}$ such that

$$\text{key}(a_{\pi_1}) \leq \text{key}(a_{\pi_2}) \leq \ldots \text{key}(a_{\pi_n}).$$

An important feature of the problem is that we are not just assuming that the input is an array of nothing but integers. Rather, the input is an array of elements with associated integer *keys*. This of great practical importance since in real-life you are rarely going to want to sort an array that contains literally nothing but integers, but likely you want to sort some data by some integer property. For example, you may have a spreadsheet and you want to sort the rows by one of the columns which contains an integer value. When you sort the rows, you of course do not only

want to sort that one column containing the integer, but you want the entire row attached to that integer to come along for the ride, otherwise the spreadsheet would be messed up and the rows would no longer contain the right values.

Note that we allow there to be duplicate keys among the elements. Recall from your previous classes that a sorting algorithm is called *stable* if it preserves the relative order of duplicate keys. That is, if $\text{key}(a_i) = \text{key}(a_j)$ and $i < j$, then a_i appears before a_j in the output. It will be of importance to us in this lecture to design algorithms that are stable.

2.1 The algorithm

When sorting an array of elements, the main question we have for each element is "where should it go?" In the comparison model when dealing with black-box comparable things, the only way we can answer this question is by comparing the element to (often many) other elements. However, when our keys are integers, we already have a pretty good idea of where they go. Element x comes after element $x-1$ and before element $x+1$ (if they exist).

Warm-up problem: Sorting distinct keys $\{1,2,\ldots,n\}$ As a warm up, lets say that the input only contains *distinct keys* in the range $\{1,2,\ldots,n\}$, which means the keys are exactly the set $\{1,2,\ldots,n\}$. In this case, we could sort them by simply creating an output array S of size n, then for each element a_i, just place a_i in $S[\text{key}(a_i)]$ directly!

Warmer-up: Sorting distinct keys in $\{0,1,\ldots,u-1\}$ If we knew the input contained only distinct keys in the range $\{0,1,\ldots,u-1\}$, we could sort them by creating an array S of size u instead of size n, and then similarly placing element a_i in slot $S[\text{key}(a_i)]$, then lastly filtering out the empty slots with a second pass over the output.

Counting Sort In general we might have duplicate keys, so we can fix this by instead storing a *list* of elements with key x at slot x. This gives us our first integer sorting algorithm.

Algorithm: Counting Sort

Suppose the input contains n elements $a_1,\ldots,a_n$ whose keys are integers in the range $0,1,\ldots,u-1$. The `key` function takes an element and returns the associated integer key.

```
function CountingSort(a : array, key : element → int) {
  let L be an array of u empty lists
  for each element x in a do {
    L[key(x)].append(x)
  }
  let out = empty list
  for each integer k in range(0, u) do {
    out.extend(L[k])
  }
  return out
}
```

The correctness follows from the fact that the elements are stored in the array in key order.

Theorem: Complexity of Counting Sort

On n elements with integer keys in $\{0, 1, \ldots, u-1\}$, Counting Sort runs in $O(n+u)$ time.

Proof. The algorithm just makes one pass over the input of length n, then one pass over the universe of keys u, and builds the output array of length n, so in total, we have an algorithm that runs in $O(n+u)$ time □

Finally, we should make the observation that Counting Sort is *stable* since it appends elements to the lists in order. This will be very important in the last section.

We wanted linear-time sorting, right? Our goal was to design an algorithm that sorts faster than a comparison sorting algorithm which we know can run in $\Theta(n \log n)$ time. Counting Sort achieves a runtime of $O(n+u)$, so how exactly does that stack up? Its not quite directly comparable since it depends on the size of u. Since our goal is to achieve linear-time sorting, we will describe the algorithm in terms of which inputs allow it to achieve this goal. So, in this case, as long as $u = O(n)$, Counting Sort runs in linear time! In other words, if our integers have at most $\log n + c$ bits for any constant c, Counting Sort is linear time. This is not too bad. If we have a data set of n elements and each has an index in $1 \ldots n$ or even $1 \ldots c\,n$ for some constant n, then we can sort on that index in linear time!

Our next goal will be to improve this and find an algorithm that can sort even larger integers still in linear time.

3 A side quest: Tuple sorting

A useful stepping stone towards our last algorithm will be a concept sometimes called *tuple sorting*. The problem is this:

Problem: Tuple sorting

Given an array of n elements where each element has a key which are equal-length tuples $(k_1, k_2, \ldots, k_d)$, we want to sort the array lexicographically by key. That is, the array should be sorted by the first element of the tuples, with ties broken by the second, ties on those broken by the third, and so.

We will describe three algorithms for this problem! All of them will work by using another ordinary sorting algorithm but in different ways.

Comparison tuple sorting The most straightforward (though not particularly useful for this lecture) algorithm for tuple sorting is to just apply any of your favourite comparison sorting algorithms (merge sort, quicksort, heapsort) that run in $O(n \log n)$ cost, comparing the tuples element by element. Note that comparing a tuple does **not** take constant time, so we need to

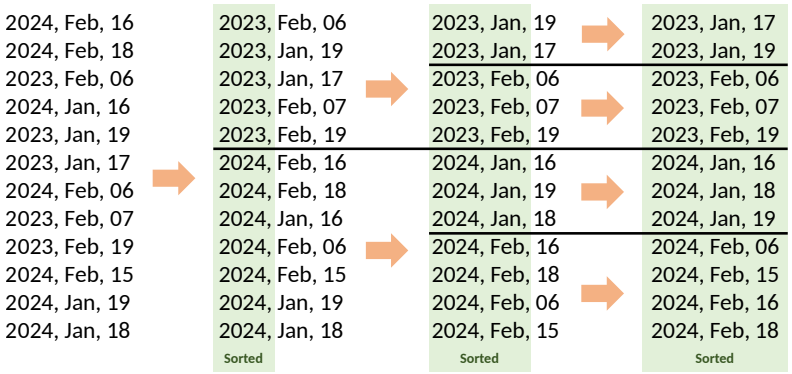
multiply the cost (number of comparisons) by the length of the tuples, i.e., d. This will give us an $O(d\,n \log n)$ algorithm. This works fine in the comparison model, but it doesn't generalize very well outside the comparison model, e.g., to tuples of integers and integer sorting.

The remaining two algorithms, instead of only performing one sort, will sort multiple times to achieve the same effect! This will also make them more generalizable and applicable outside the comparison model since each sort will only apply to one element of the tuple.

3.1 Top-down tuple sorting

A natural algorithm for tuple sorting is to essentially follow the definition. First, we can sort, using any sorting algorithm we like, using the first tuple element as the key. The array is now sorted correctly *except* that keys with equal first tuple elements are not tie-broken by the second tuple element yet. To resolve that, recursively sort each subarray of equal first elements, this time using the second tuple element as the key, and so on.

As an example, consider sorting a set of dates which are represented as (Year, Month, Day) tuples. A top-down tuple sort will first sort them by year, then recursively sort the dates that have equal years by their month, and then finally recursively sort the dates with equal years and months by their day.



The nice thing about this algorithm is that it can be applied using any sorting algorithm to perform each step, as long as that sorting algorithm can sort the individual elements of the tuples, e.g., we could use counting sort if every element of the tuples are integers!

3.2 Bottom-up tuple sorting

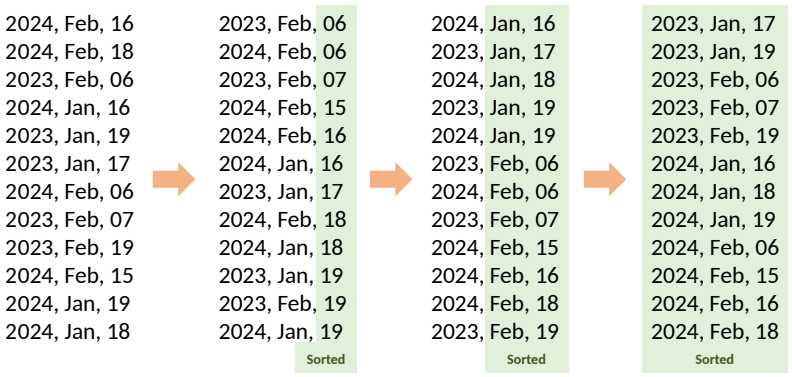
Like most recursive algorithms, we can also find a non-recursive alternative. What if instead of recursively sorting each subarray of equal tuple elements, we just sorted the entire array one tuple element at a time? Say we first sort the entire array using the first tuple element as the key, then the second, and so on. What would happen? Well, the array wouldn't actually be in sorted order because it would be sorted by the final tuple elements, not the first!

However, this algorithm almost works, with just one minor adjustment. The tuple element that we sort with respect to last is the one that ends up sorted, so we should actually sort in *reverse order* of the tuple elements, i.e., sort with respect to the final tuple element, then the second

last, and so on until we sort with respect to the first tuple element. This makes intuitive sense because the final sort is going to dictate that the elements are ordered by the first tuple element, which is exactly what we want.

Now the important observation: as long as the sorting algorithm that we use at each step is *stable*, we will end up with the correct answer. This is because before we sort on the first tuple element, the array is already sorted correctly with respect to the second tuple element because of the previous sort, so by using a stable sort, all ties between equal first tuple elements will be correctly tie-broken on the second tuple element! Applying this reasoning inductively should convince us that the array will be correctly sorted lexicographically with respect to the tuples!

Lets see the date sorting example again, this time using the bottom-up approach. Notice that the algorithm pretty much does the same thing but in reverse! First it sorts the days correctly, then it sorts the months so that the (Month, Day) pairs are in sorted order, then finally it sorts the years to bring everything into the correct order.



The key difference is that unlike the top-down approach, each round sorts the whole array, rather than recursively subdividing the array into smaller pieces that are sorted separately.

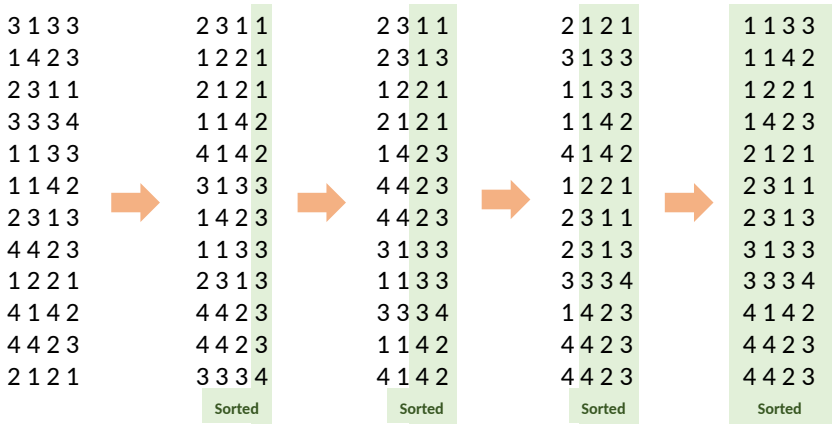
4 Sorting bigger integers: Radix Sort

We now have the ingredients to derive our last and best integer sorting algorithm. Counting Sort is great for sorting small integers but falls over once u becomes too large. However, what if we were to use *tuple sorting* instead on a tuple of small integers? As long as the tuple elements are small that sounds like it might be efficient! So the question is how can we convert a bigger integer into a tuple of smaller integers such that sorting on the tuples is equivalent to sorting on the original values? It turns out that we definitely already know the answer to this question, because it is just how we write down numbers every day! We can split numbers into their *digits*!

Key Idea: Integer sorting is tuple sorting by digits

Sorting integers is equivalent to tuple-sorting them by their digits! If a number has a greater first digit than another, then it is greater. If two numbers have equal first digits, then they are tie-broken by their second digit and so on!

Here's the idea in picture form, just like bottom-up tuple sorting from a moment ago. We first sort the numbers by the least-significant digit, then by the second-least, and so on, until finally sorting by the most significant digit. This algorithm is called bottom-up Radix Sort or *Least-Significant-Digit (LSD) Radix Sort*, since it sorts in order from least to most significant.



Since it is stable and the digits are small integers, we use Counting Sort for each iteration!

Algorithm: LSD Radix Sort

Suppose the input contains n elements $a_1, \ldots, a_n$ whose keys are integers with `num_digits` digits, and we have a function `Digit(x, i)` which extracts the i^{th} digit of x.

```
function RadixSort(a : array, key : element → int) {
  let out = copy of a
  for each i in range(0, num_digits) do {
    out = CountingSort(out, key = x → Digit(key(x), num_digits - i))
  }
  return out
}
```

We could similarly implement a recursive top-down / MSD Radix Sort as well, but we will stick with this for now since they have the same complexity. In practice it would be more efficient to allocate the `out` array just once and reuse it (e.g., swapping it with `a` each iteration) for every iteration of Counting Sort rather than constantly allocating new arrays each time.

Analysis The last step is to analyze the algorithm. How fast is it? Well, it depends on the number of digits, right? If our numbers are again from a universe $\{0, 1, \ldots, u-1\}$, then they have $d = \log_b u$ digits when written in base b. Each digit will be a smaller integer from $\{0, 1, \ldots, b-1\}$. The algorithm performs d iterations, each of which performs a counting sort. Since the digits are in base b, Counting Sort takes $O(n+b)$ time. Therefore the total time for Radix Sort is

$$O((n+b)\log_b u).$$

Initially this doesn't look that great. If we pick any constant base b, then even if $u = O(n)$, the runtime becomes $O(n \log n)$, which is the same as comparison sorting and actually worse than

Counting Sort! How can we make this better? Well, the higher the base, the fewer iterations the algorithm needs, so lets make it even higher! As long as we do not cause the Counting Sort to take more than linear time, we should be fine, so since the Counting Sort takes $O(n+b)$ time, our base can go up to n without trouble! Intuitively this makes sense because Counting Sort was good until the universe became bigger than linear in n, so we should pick that as the base for Radix Sort to get the most value out of each iteration of Counting Sort.

Theorem: Complexity of Radix Sort

On n elements with integer keys in $\{0,1,\ldots,u-1\}$, Radix Sort runs in $O(n \log_n u)$ time.

This is a little hard to interpret and its not immediately obvious whether this is good, but the claim is that this is now good and can sort much larger integers than Counting Sort could!

Corollary: Radix Sort can sort bigger integers

Radix Sort can sort n elements with integers keys in the range $\{0,1,\ldots,O(n^c)\}$ for any constant c, i.e., any integer keys that are *polynomial size* in n, **in linear time**.

Proof. Let $u = O(n^c)$, so $\lg_n u = O(\lg_n n^c) = O(c) = O(1)$ and hence the complexity is $O(n)$. □

Wow, that's a big improvement! We have gone from being able to only sort integer keys that were *linear* in n, i.e., keys up to $O(n)$, which is quite restrictive, to being able to sort any keys that are *polynomial* in n, i.e., keys up to $O(n^c)$ for any constant c.

Remark: Integer sorting is still an open problem!

One cool thing about comparison sorting is that it is (asymptotically) a solved problem. We know that $\Omega(n \log n)$ is a lower bound, but we also have algorithms that run in $O(n \log n)$ cost, so those algorithms are optimal up to constant factors.

The same however is **not true** for integer sorting! The algorithms we learned today can sort integers that are up to polynomial size in n, but what about sorting integers without any restriction on the size? This is still an open research problem. The best algorithms that have been discovered run in $O(n \log\log n)$ time (deterministically), or in $O(n\sqrt{\log\log n})$ expected time. We don't however know any lower bound for integer sorting other than the trivial $\Omega(n)$ lower bound! So, we still don't know whether there exists a linear-time sorting algorithm for integers that works regardless of their size, or whether there is a lower bound that shows that this is not possible. Either could be true!

Exercises: Integer Sorting

Problem 1. We wrote pseudocode that performs a bottom-up (LSD) Radix Sort. Write similar pseudocode for a top-down Most-Significant-Digit (MSD) Radix Sort instead.

Problem 2. Argue that the complexity of MSD Radix Sort is the same as LSD / bottom-up.