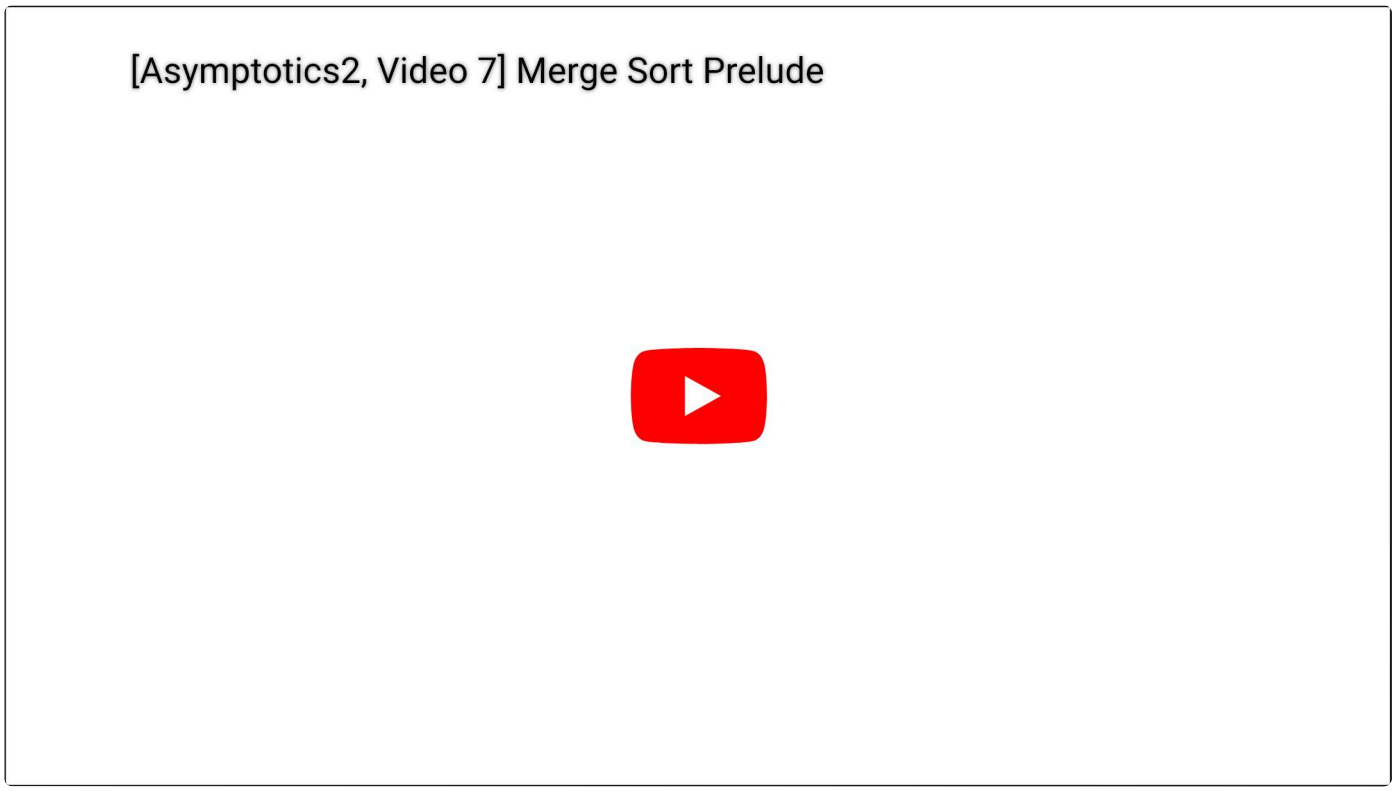
15.4 Mergesort

In our last example, we'll analyze merge sort, another cool sorting algorithm.



Mergesort basics: merging two sorted lists.

First, let's remind ourselves of selection sort, which we will initially use as a building block for merge sort.

Selection sort works off two basic steps:

- Find the smallest item among the unsorted items, move it to the front, and 'fix' it in place.
- Sort the remaining unsorted/unfixed items using selection sort.

If we analyze selection sort, we see that its runtime is $\Theta(N^2)$.

Exercise: To convince yourself that selection sort has $\Theta(N^2)$ runtime, work through the geometric approach (try drawing out the state of the list at every sort call), or count the

operations.

Let's introduce one other idea here: **arbitrary units of time**. While the exact time something will take will depend on the machine, on the particular operations, etc., we can get a general sense of time through our arbitrary units (AU).

If we run an $N = 6$ selection sort, and the runtime is of order N^2, it will take ~36 AU to run. If $N = 64$, it'll take ~2048 AU to run. Now we don't know if that's 2048 nanoseconds, or seconds, or years, but we can get a relative sense of the time needed for each size of N .

Hold onto this thought for later analysis.

Now that we have selection sort, let's talk about **merging.**

Say we have two **sorted** arrays that we want to combine into a single big sorted array. We could append one to the other, and then re-sort it, but that doesn't make use of the fact that each individual array is already sorted. How can we use this to our advantage?

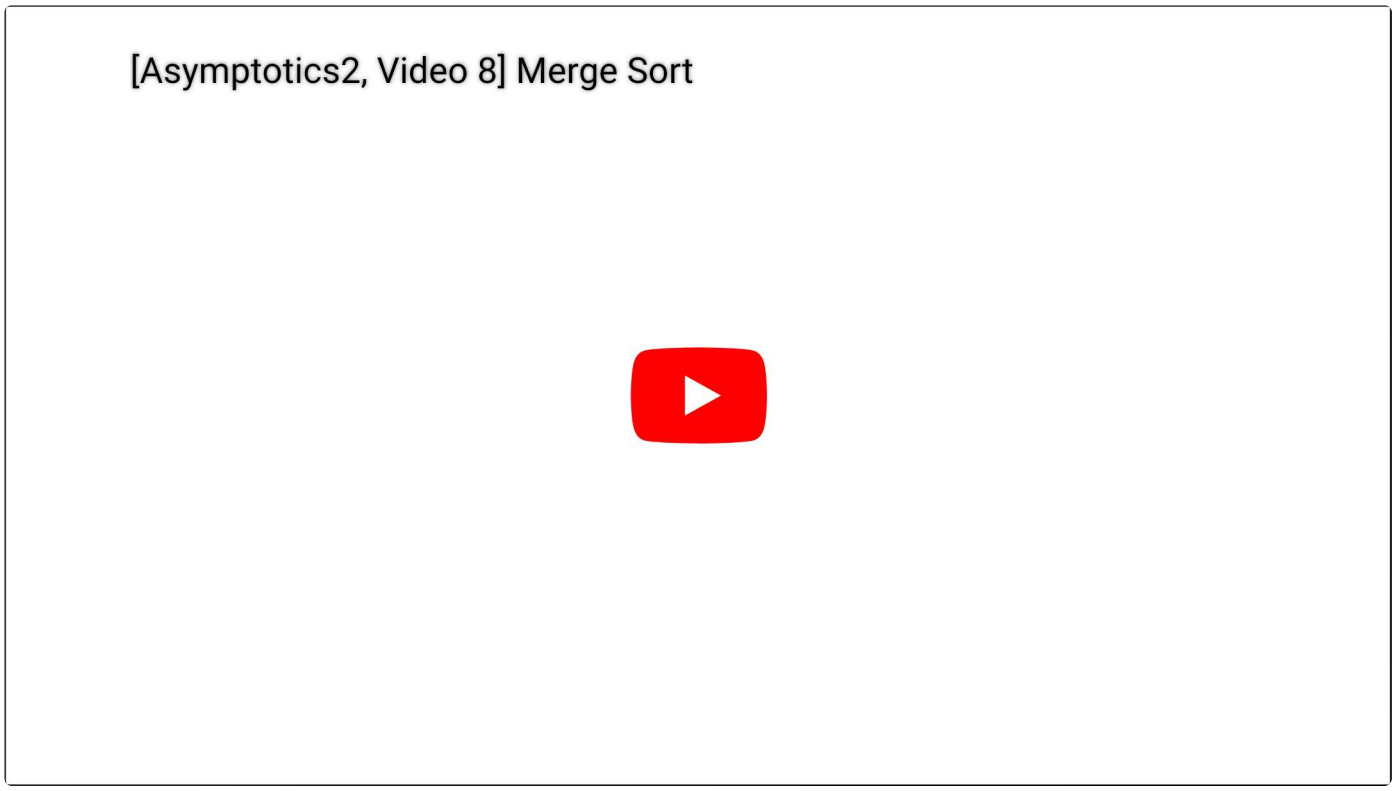
It turns out, we can merge them more quickly using the sorted property. The smallest element must be at the start of one of the two lists. So let's compare those, and put the smallest element at the start of our new list.

Now, the next smallest element has to be at the new start of one of the two lists. We can continue comparing the first two elements and moving the smallest into place until one list is empty, then copy the rest of the other list over into the end of the new list.

To see an animation of this idea, go here.

What is the runtime of the merge operation? We can use the number of "write" operations to the new list as our cost model, and count the operations. Since we have to write each element of each list only once, the runtime is $\Theta(N)$.

Selection sort is slow, and merging is fast. How do we combine these to make sorting faster?



A closer look at Mergesort

We noticed earlier that doing selection sort on an $N = 64$ list will take ~2048 AU. But if we sort a list half that big, $N = 32$, it only takes ~512 AU. That's more than twice as fast! So making the arrays we sort smaller has big time savings.

Having two sorted arrays is a good step, but we need to put them together. Luckily, we have merge. Merge, being of linear runtime, only takes ~64 AU. So in total, splitting it in half, sorting, then merging, only takes 512 + 512 + 64 = 1088 AU. Faster than selection sorting the whole array. But how much faster?

Now, AUs aren't real units, but they're sometimes easier and more intuitive than looking at the runtime. The runtime for our split-in-half-then-merge-them sort is $N + 2(\frac{N}{2})^2$, which is about half of N^2 for selection sort. However, they are still both $\Theta(N^2)$.

What if we halved the arrays again? Will it get better? Yes! If we do two layers of merges, starting with lists of size $\frac{N}{4}$, the total time will be ~640 AU.

Exercise: Show why the time is ~640AU by calculating the time to sort each sub-list and then merge them into one array.

What if we halved it again? And again? And again?

Eventually we'll reach lists of size 1. At that point, we don't even have to use selection sort, because a list with one element is already sorted.

This is the essence of **merge sort:**

- If the list is size 1, return. Otherwise:
- Mergesort the left half
- Mergesort the right half
- Merge the results

So what's the running time of **merge sort**?

We know merge itself is order N, so we can start by looking at each layer of merging:

- To get the top layer: merge ~64 elements = 64 AU
- Second layer: merge ~32 elements, twice = 64 AU
- Third layer: ~16*4 = 64 AU
- ...

Overall runtime in AU is ~64*k, where k is the number of layers. Here, $k = log_2(64) = 6$, so the overall cost of mergesort is ~384 AU.

Now, we saw earlier that splitting up more layers was faster, but still order N^2. Is merge sort faster than N^2?

Yes!
Mergesort has worst case runtime $\Theta(N * log(N))$.

- The top level takes ~N AU.
- Next level takes ~N/2 + ~N/2 = ~N.
- One more level down: ~N/4 + ~N/4 + ~N/4 + ~N/4 = ~N.

Thus, total runtime is ~Nk, where k is the number of levels.

How many levels are there? We split the array until it is length 1, so $k = log_2(N)$. Thus the overall runtime is $\Theta(N * log(N))$.

Exercise: Use exact counts to argue for $\Theta(N * log(N))$. Account for cases where we cannot divide the list perfectly in half.

So is $\Theta(N * log(N))$ actually better than $\Theta(N^2)$? Yes! It turns out $\Theta(N * log(N))$ is not much slower than linear time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

timing_table_for_runtimes

Previous
15.3 Binary Search

Next
15.5 Summary

Last updated 1 year ago