



# Making Sense of Merge Sort [Part 1]



Vaidehi Joshi · [Follow](#)

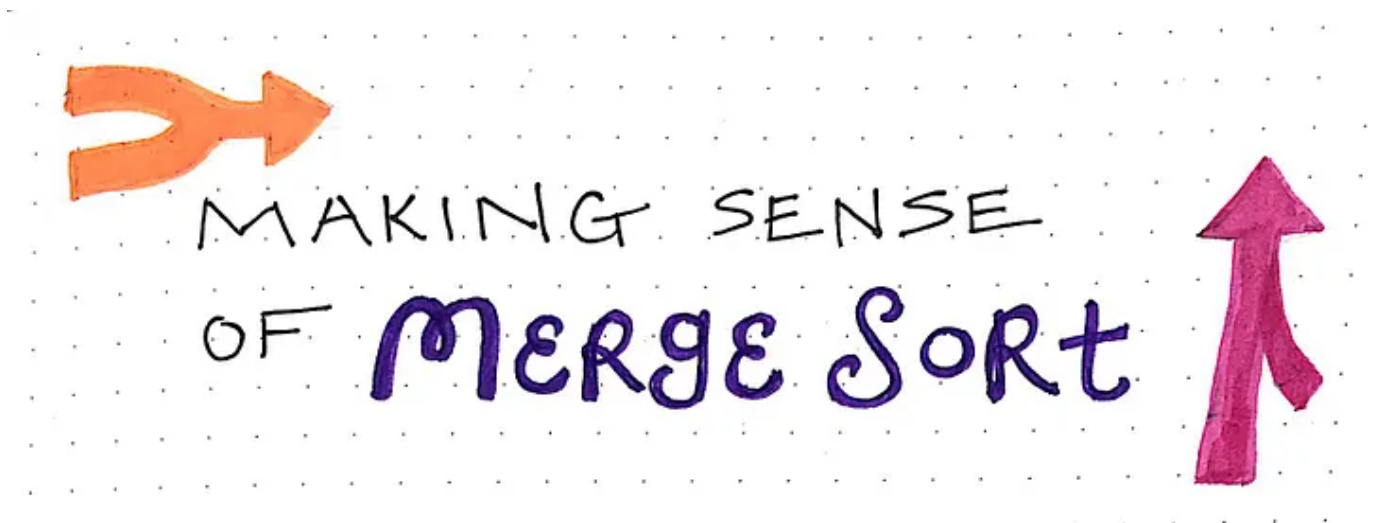
Published in [basecs](#) · 11 min read · Jun 5, 2017



2K



6



Making sense of merge sort

If you've been reading this series sequentially, then there's a good chance that over the course of the past few weeks, you've thought more about sorting things than you've ever thought about before! At least, that has certainly been the case for me. I haven't started dreaming about sorting algorithms just yet, but I'm expecting that it might happen soon.

Well, guess what? We're not quite halfway through this series yet — and we're not quite halfway through sorting algorithms yet, either! However, we *are* at a turning point our algorithm adventure. So far, we've talked about some of the most common — and sometimes thought of as the more “simple” — sorting algorithms. We've covered selection sort, bubble sort, and insertion sort. If you take a closer look at these algorithms, you might notice a pattern: they're all pretty slow. In fact, all of the sorting algorithms that we've explored thus far have had one thing in common: they're all pretty inefficient! Each of them, despite their little quirks and differences, have a *quadratic* running time; in other words, using Big O notation, we could say that their running time is  $O(n^2)$ .

This commonality was completely intentional—surprise! The order in which we're learning these topics is actually pretty important: we're covering sorting algorithms based on their *time complexity*. So far, we've covered algorithms with a quadratic runtime complexity, but from this point forwards, we'll be looking at algorithms that are significantly faster and more efficient. And we'll start off with one of the most fun (albeit a little more complex) sorting algorithms there is!

Don't worry, we're going to break it down, step by step. I suppose that step one is telling you what this algorithm is called! It's time for you to meet my new friend: merge sort.

## **Divide and conquer algorithms**

You might have heard merge sort discussed or referenced in the context of a technical interview, or a computer science textbook. Most CS courses spend a decent amount of time covering this topic, and for some reason or another, this particular algorithm seems to pop up quite a lot.

Interestingly, however, merge sort was only invented (discovered?) in 1945, by a mathematician named John von Neuman. In the grand scheme of things, merge sort is still a fairly new algorithmic approach to sorting. But hang on a second — we still don't know what it is yet!

Alright, time for a definition. The *merge sort algorithm* is a sorting algorithm that sorts a collection by breaking it into half. It then sorts those two halves, and then merges them together, in order to form one, completely sorted collection.



A merge sort algorithm splits an unsorted collection into 2 halves; it sorts the 2 halves, and then merges them together to form one, sorted collection — all of this, using recursion

Merge sort: a definition

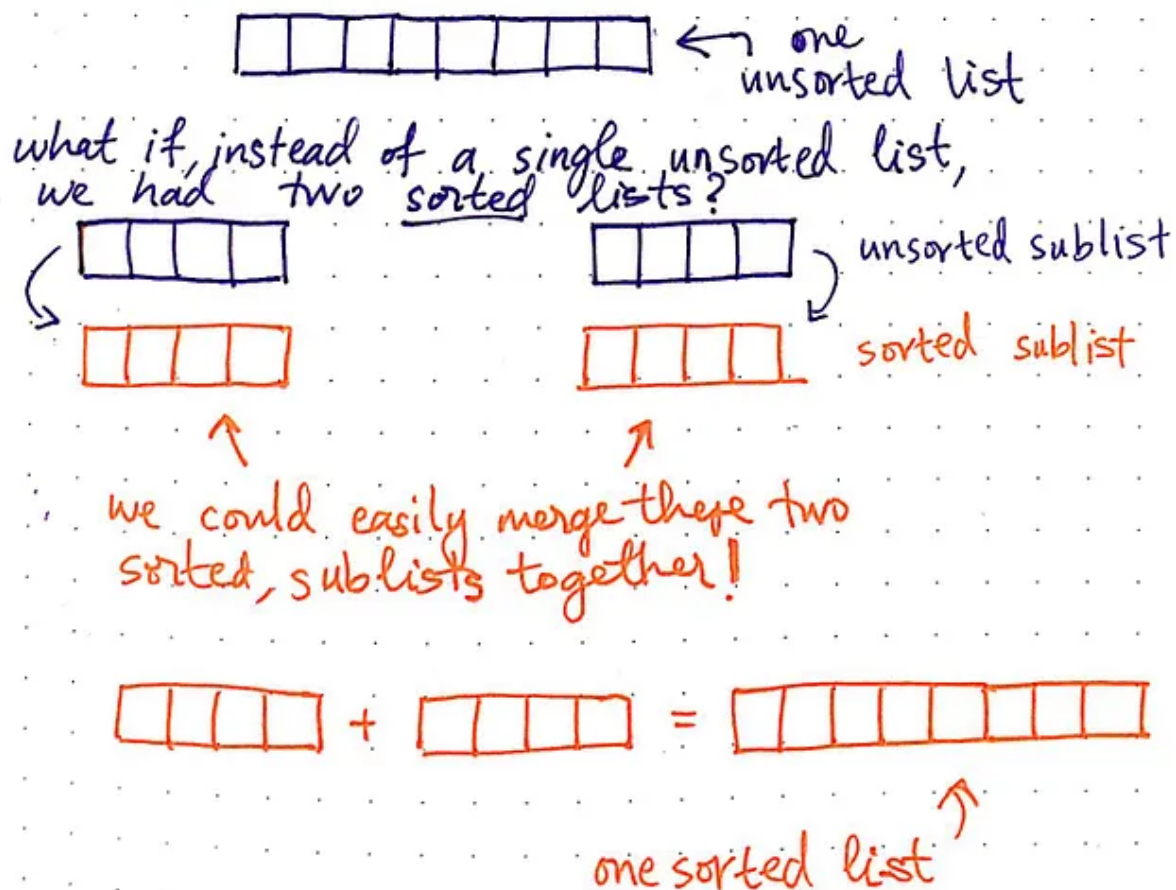
And, in most implementations of merge sort, it does all of this using *recursion*. But hold that thought for now — we'll come back to recursion in just a moment.

The basic idea behind merge sort is this: it tends to be a lot easier to sort two smaller, sorted lists rather than sorting a single large, unsorted one.

We've already seen how sorting through one large, unsorted list can end up being slow. If we think back to selection sort, we ran into the issue of having to iterate through the list multiple times, selecting the smallest element and marking it as "sorted". With bubble sort, we again had to pass through the list many, many times; each time, we compared just two elements at a time, and then swapped them. Bubble sort was more obviously slow since we had to make so many swaps! And then there was insertion sort, which was kind of acceptable if our list was already mostly sorted. But if it wasn't, then we basically were forced to iterate through a list and, one item at a time, slowly pull out the smallest element and insert it into its correct spot in a sorted array.

This is where merge sort fundamentally differs from all of these other sorting algorithms. Let's take a look at an example to help illustrate this.

# Merge Sort Theory :



Merge sort theory: how does it even?

In the image here, we have a single, unsorted list. Conceptually, merge sort asserts that instead of a one unsorted list, it's a lot easier to sort and join together two *sorted* lists. The idea is that if we could somehow magically end up with two sorted halves, then we could very easily merge those two sorted sublists together. Ultimately, if we did our merging in a smart, efficient way, we could end up with one sorted list at the end of it all.

Hopefully, at this point, you're wondering how on earth merge sort can just "magically" split and sort two halves of our list.

Hang on for a second — we're programmers! Intrinsically we know that, at the end of the day, there really is no magic at play here. There's something else going on under the hood, and it's probably an abstraction of something else.

In the case of merge sort, that abstraction is something called *divide and conquer* (sometimes referred to as *d&c*). The divide and conquer technique is actually an *algorithm design paradigm*, which is really just a fancy way of saying that it's a design pattern that lots of algorithms use! In fact, we've already seen this exactly paradigm before, way back when we were first learning about the binary search algorithm.

So what does the divide and conquer paradigm entail, exactly? Well, for starters, an algorithm that uses the divide and conquer strategy is one that divides the problem into smaller subproblems. In other words, *it breaks down the problem into simpler versions of itself*.

The basic steps of a d&c algorithm can be boiled down to these three steps:

1. **Divide** and break up the problem into the smallest possible “subproblem”, of the exact same type.
2. **Conquer** and tackle the smallest subproblems first. Once you've figured out a solution that works, use that exact same technique to solve the larger subproblems — in other words, solve the subproblems recursively.
3. **Combine** the answers and build up the smaller subproblems until you finally end up applying the same solution to the larger, more complicated problem that you started off with!

# Divide + Conquer!

- ➔ A divide and conquer algorithm divides a problem into simpler versions of itself.
- ➔ By breaking down a problem into smaller parts, they become easier to solve. Usually, the solution for the smaller sub-problems can be applied to the larger, complicated one.
- ➔ Conquering the large problem using the same solution is what makes d+c recursive.

The rules of divide and conquer algorithms

The crux of divide and conquer algorithms stems from the fact that it's a lot easier to solve a complicated problem if you can figure out how to split it up into smaller pieces. By breaking down a problem into its individual parts, the problem becomes a whole lot easier to solve. And usually, once you figure out how to solve a "subproblem", then you can apply that *exact* solution to the larger problem. This methodology is exactly what makes recursion the tool of choice for d&c algorithms, and it's the very reason that merge sort is a great example of recursive solutions.

## Spotting recursion in the (algorithm) wild

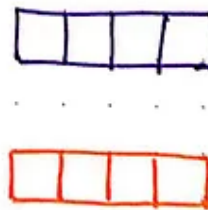
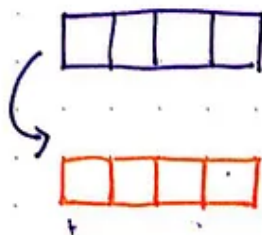
Understanding divide and conquer in theory is one thing, but its usefulness tends to be far more obvious if we can see it in action. Let's take a look at



how we can use recursion to divide and conquer the illusive merge sort algorithm!

## Merge Sort in Practice:

How can we sort our sublists before we go about merging them...?



how do we turn the unsorted halves into sorted sublists?

We can use divide & conquer, and apply the merge sort algorithm on itself — recursively!

Merge sort: in practice

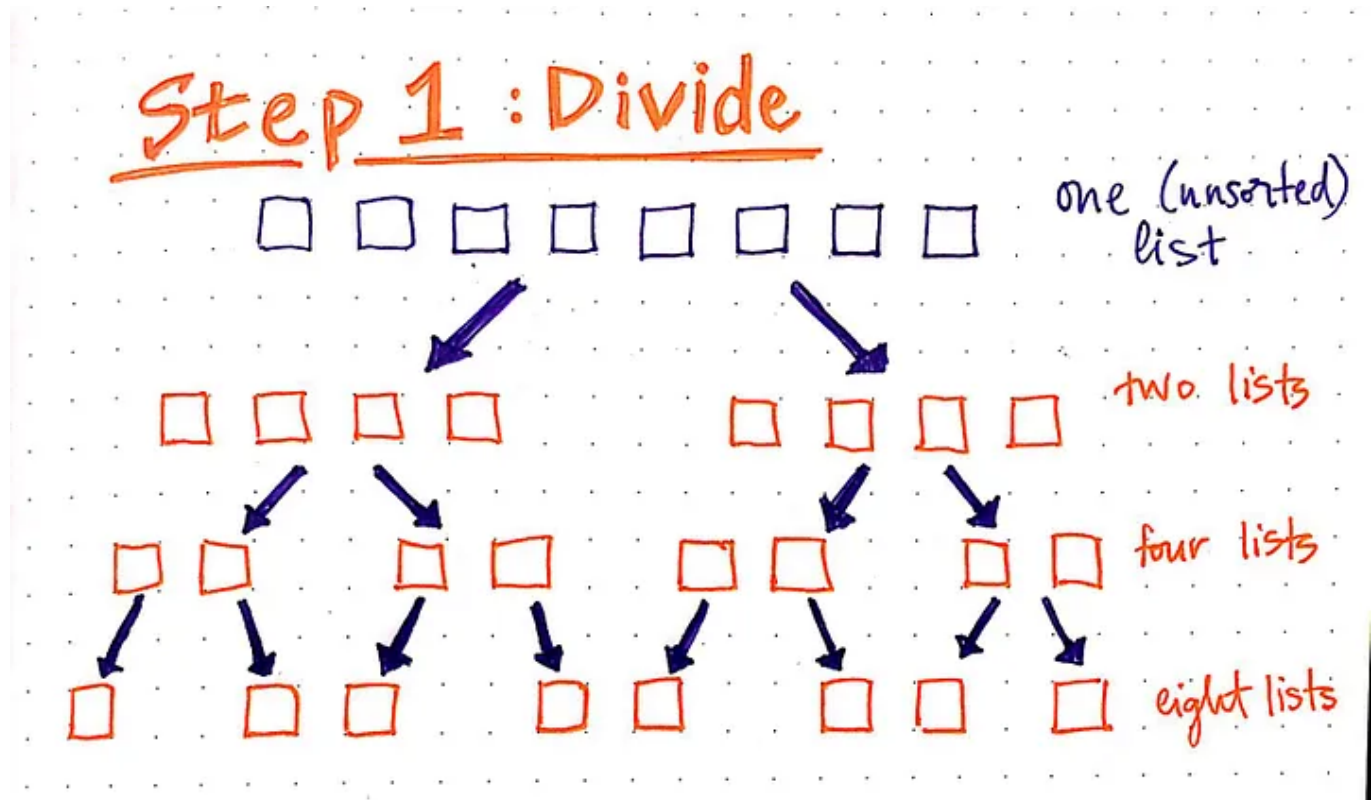
In theory, we understand that merge sort splits up an unsorted list, sorts the two halves, and merges them together.

But how does this algorithm really work in practice?



Well, now that we know what exactly that “magic” is that goes on behind the scenes — that is to say, the abstraction that’s responsible for sorting the two sublist halves—we can finally try and understand how it works. It’s time for us to apply the divide and conquer paradigm to merge sort and figure out what’s actually going on inside this algorithm!

We’ll start with a simple example, and we won’t actually worry about sorting any values just yet. For now, let’s try to understand how the divide and conquer methodology comes into play. We know that first need to divide and break up the problem into the smallest possible “subproblem”, of the exact same type. The smallest possible “subproblem” in our situation is our base case — the point at which we’ve basically solved our problem. In terms of sorting items, the base case is a sorted list. So, we can divide our large problem (and our unsorted list) into it’s smallest possible pieces.



Step 1: Dividing

In the example above, we just continue to divide our problem into smaller subproblems. We start off with one, unsorted list. Then we break that up into two, unsorted lists. We can divide that even further, into four lists. And then we can break that down again, into eight lists. At this point, we've got eight lists, and each list has only *one* item in it.

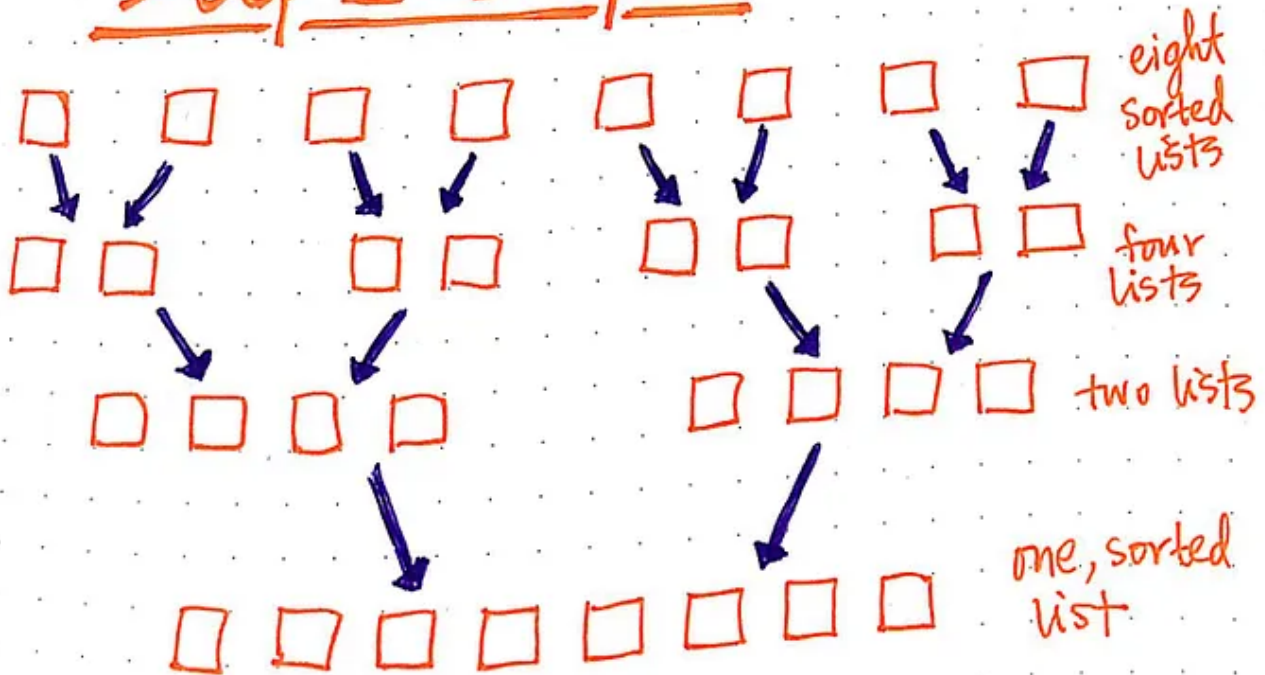
We can't divide these any further, can we? So, we've arrived at our smallest possible subproblem, and our base case: a sorted list, with one single item in it. You might remember from previous sorting algorithms that a list with only one item in it is, by definition, considered sorted. This is because there's literally nothing else — no other item — to compare it to!

Okay, now we have eight *sorted* lists, and each one as only one item in it.

According to divide and conquer, we know that the next step is to tackle our smallest subproblems first. Once we can determine a solution that works, we'll apply that same solution to solve the larger subproblems — in other words, we'll use recursion.

So, let's start by combining (merging) two items together at a time. Since we have eight sorted lists, let's combine each list with its neighbor. When we do this, we'll put the items in the correct, sorted order.

## Step 2: Conquer



Step 2: Conquering!

Nice! That was pretty cool. We took our eight, sorted lists, and merged them together to create four sorted lists. Then we took *those* four lists, and merged them together to form two sorted lists.

Oh, wait — two sorted lists? Hang on...that sounds familiar. I think we just figured out the “magic” behind merge sort! How rad is that?!

Now, since we have two sorted lists, we can finish up our merge sort by *merging* them together.

## Why does this work??

- ➔ A list with a single item in it is always going to be considered "sorted".
- ➔ Once we have at least two sorted lists, we can merge them together to create one single, sorted list.
- ➔ By splitting up one large, unsorted list down into its individual parts, we can then begin merging those elements together, effectively reconstructing our list—but now, in sorted form.

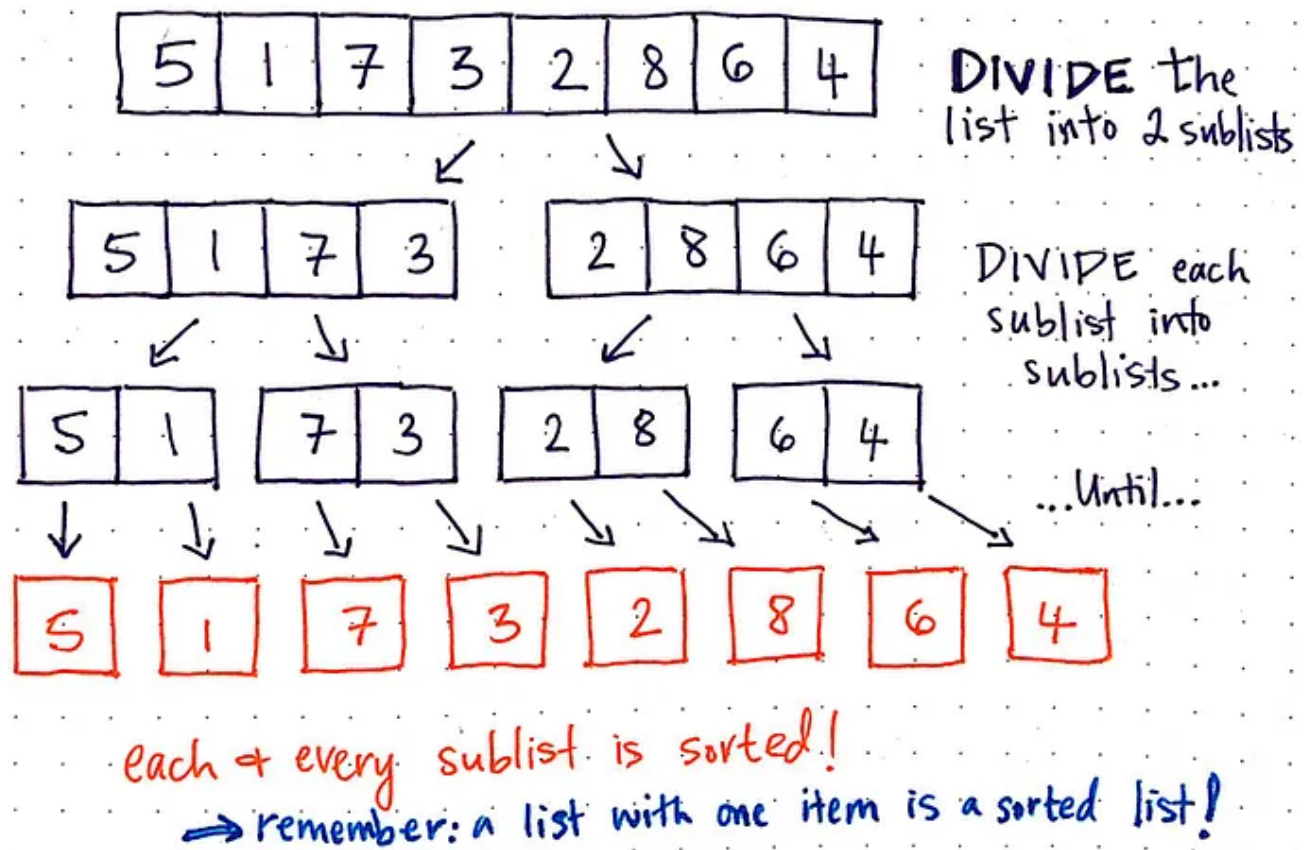
How does merge sort work with recursion?

There are a few reasons that divide and conquer actually works so well in implementing merge sort. For starters, the fact that a list with a single item is, by definition, a "sorted" list is what makes it easy for us to know when we've hit our smallest possible "subproblem". When this algorithm is implemented in code, this ends up being the base case for determining when the recursion should end. We've talked about recursion before in the context of binary trees; in that situation, the base case is a single leaf node — in other words, when you can't break down a subtree into any smaller possible parts. The same idea applies here: when we can't break down our list into any smaller possible parts, and when we only have one item that is sorted, we've hit our recursive base case.

Another reason that divide and conquer works here is once we know how to merge two items together and have figured out the logic behind that, we basically can just keep reusing that same logic and continue applying it to every built-up sublist that we merge together.

This is exactly what makes recursion so powerful: once we've figured out how to solve the problem of merging two lists together, it doesn't matter if the list has one element, or a hundred — we can reuse the same logic in both scenarios.

Let's take a look at one more example — this time, we'll use actual numbers that we're trying to sort. In the drawing below, we have an unsorted list with eight numbers. We can start by dividing the unsorted collection into two sublists. We'll just continue dividing until we hit our base case.



Merge sort in action: step 1 — dividing

Okay, now we're down to the smallest possible subproblem: eight lists, and each has one, sorted item within it. Now, we just need to merge two lists together. We'll merge each sorted list together with its neighbor. When we merge them, we'll check the first item in each list, and combine the two lists together so that every element is in the correct, sorted order.

Easy-peasy! We got this.



5 1 7 3 2 8 6 4

merge each sorted list together with its neighbor — maintaining sorted order

1 5 3 7 2 8 4 6

1 3 5 7 2 4 6 8

continue merging sublists until...

1 2 3 4 5 6 7 8

there is only one part left:  
the sorted list, merged together!

Merge sort in action: step 2 — merging

Great! Once we started merging two lists together, we didn't have to think too much more when it came to merging those two sorted sublists, did we? We used the same technique to merge lists with four items as we did when we merged lists with only one item.



Okay, illustrations are great — but what would this look like in code? How would we even be able to spot the recursive part of a merge sort algorithm if we saw one in the wild?

## Recursion at work

We already know that a recursive algorithm is one that conceptually uses *the same solution* to solve a problem. When it comes to code, this can effectively be translated to a function that *calls itself*.

In the code below — which is adapted from Rosetta Code's JavaScript implementation of merge sort — we can see that the `mergeSort` function actually *calls itself*.

```

1  function mergeSort(array) {
2      // Determine the size of the input array.
3      var arraySize = array.length;
4
5      // If the array being passed in has only one element
6      // within it, it is considered to be a sorted array.
7      if (arraySize === 1) {
8          return;
9      }
10
11     // If array contains more than one element,
12     // split it into two parts (left and right arrays).
13     var midpoint = Math.floor(arraySize / 2);
14     var leftArray = array.slice(0, midpoint);
15     var rightArray = array.slice(midpoint);
16
17     // Recursively call mergeSort() on
18     // leftArray and rightArray sublists.
19     mergeSort(leftArray);
20     mergeSort(rightArray);
21
22     // After the mergeSort functions above finish executing,
23     // merge the sorted leftArray and rightArray together.
24     merge(leftArray, rightArray, array);
25
26     // Return the fully sorted array.
27     return array;
28 }
29
30 function merge(leftArray, rightArray, array) {
31     var index = 0;
32
33     while (leftArray.length && rightArray.length) {
34         console.log('array is: ', array);
35         if (rightArray[0] < leftArray[0]) {
36             array[index++] = rightArray.shift();
37         } else {

```

Open in app ↗



🔍 Search

✍ Write



```

42 while (leftArray.length) {
43     console.log('left array is: ', leftArray);
44     array[index++] = leftArray.shift();
45 }

```

```

45     }
46
47     while (rightArray.length) {
48         console.log('right array is: ', rightArray);
49         array[index++] = rightArray.shift();
50     }
51
52     console.log('** end of merge function ** array is: ', array);
53 }

```

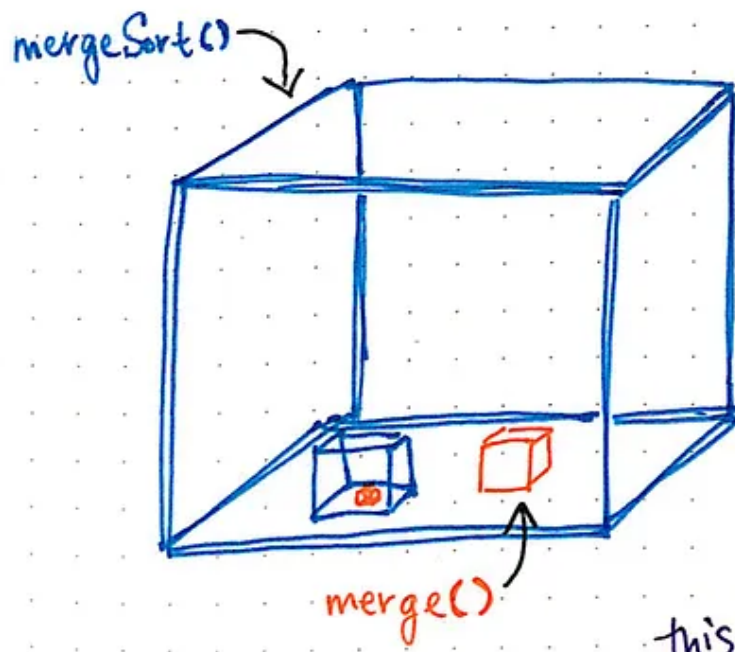
merge\_sort.js hosted with ❤ by GitHub

[view raw](#)

There are a few things going on here, and we won't dive into all of them today (don't worry, we'll come back to it next week!). For now, let's just look at what makes this algorithm recursive in nature. We can see that we're taking in an input array, and splitting it as close as we can to the center — in this case, we call it `midpoint`.

Then, we take those two halves (`leftArray` and `rightArray`), and we actually pass those in as the new input arrays to the *internal* calls to `mergeSort`. Guess what? This is recursion in action!

We can think of it recursion kind of like Russian babushka dolls — or like boxes with smaller boxes within them.



We can think of a `mergeSort` function as a function with two parts inside it:

1) a `merge` function

2) a `mergeSort` function


↑  
this is recursion in action!

How recursion works inside of an implementation of a `mergeSort` function

A `mergeSort` function ultimately has two functions inside of it:

1. a `merge` function, which actually combines two lists together and sorts them in the correct order
2. and a `mergeSort` function, which will continue to split the input array again and again, recursively, and will also call `merge` again and again, recursively.

Indeed, it is *because* merge sort is implemented recursively that makes it faster than the other algorithms we've looked at thus far.



Because merge sort is implemented recursively, we can sort multiple elements/sections of the list at a time (rather than one item at a time, in an iterative sort).

Recursive vs iterative solutions

But why is this? And just how *much* faster is merge sort, exactly? Well, you'll get the answer to both of those questions next week! In part 2 of this series, we'll look at the runtime complexity of merge sort, how this recursion actually makes it more efficient, and how merge sort stacks up against other algorithms. Until then — happy merging!

## Resources

Merge sort comes up quite a lot in computer science curriculums, technical interviews, and even in textbooks. There's a wide array of resources for this particular algorithm simply because it's one of the more tricky sorting algorithms to understand. If some of this still doesn't make sense to you — or if you'd like to see more examples — try taking a look at these different resources below.

1. [Binary Search & Merge Sort](#), Department of Computer Science, Carnegie Mellon University
2. [Mergesort](#), Ian Foster
3. [Merge Sort](#), Harvard CS50

4. Merge sort algorithm, mycodeschool

5. External Sorting, GeeksForGeeks

JavaScript

Algorithms

Sorting Algorithms

Computer Science

Code



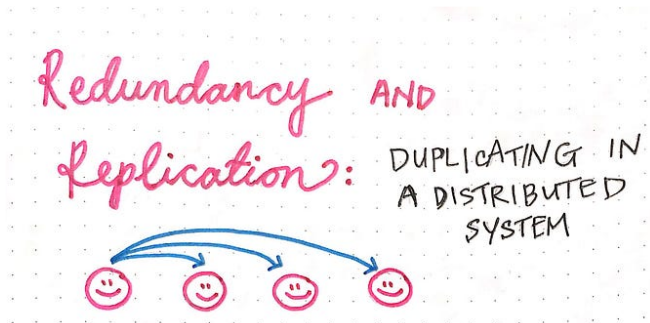
Written by Vaidehi Joshi

Follow

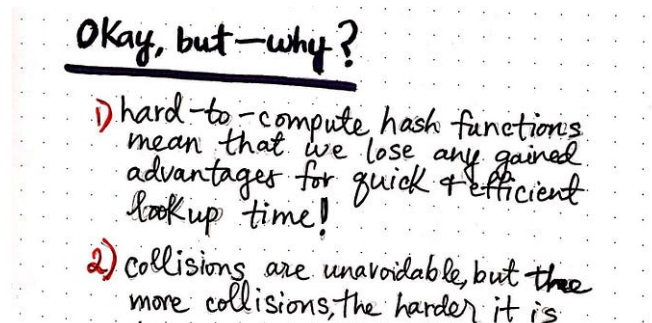
29K Followers · Editor for basecs

Writing words, writing code. Sometimes doing both at once.

More from Vaidehi Joshi and basecs



Vaidehi Joshi in basecs



Vaidehi Joshi in basecs