**Sorting Algorithms:**

**Merge Sort:**

**Reference Link:**

Making Sense of Merge Sort [Part 1] - <https://medium.com/basecs/making-sense-of-merge-sort-part-1-49649a143478>

Making Sense of Merge Sort [Part 2] - <https://medium.com/basecs/making-sense-of-merge-sort-part-2-be8706453209#:~:text=This%20algorithm%20uses%20a%20temporary,sublists%E2%80%9D%20into%20the%20original%20array>.

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.

* 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(n²).**

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!

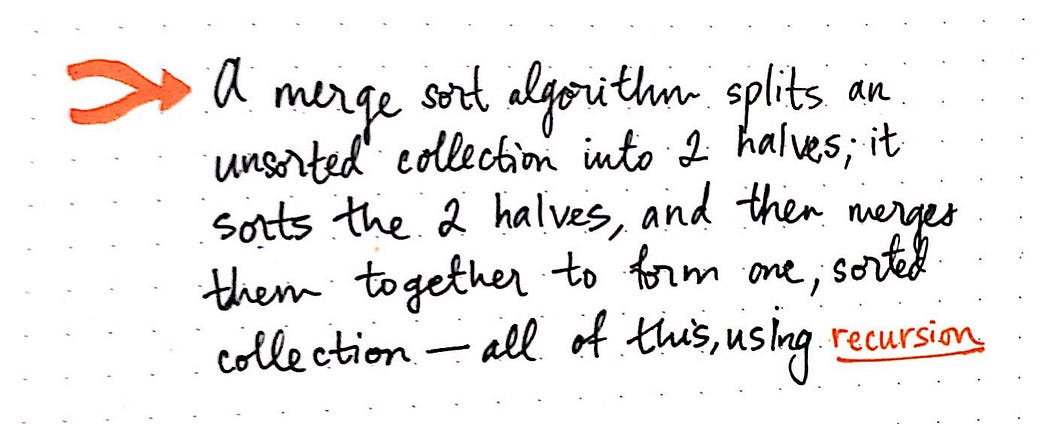
Don’t worry, we’re going to break it down, step by step. I suppose that step one is telling you what this algorithm is called! **It’s time for you to meet my new friend: merge sort.**

**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.**

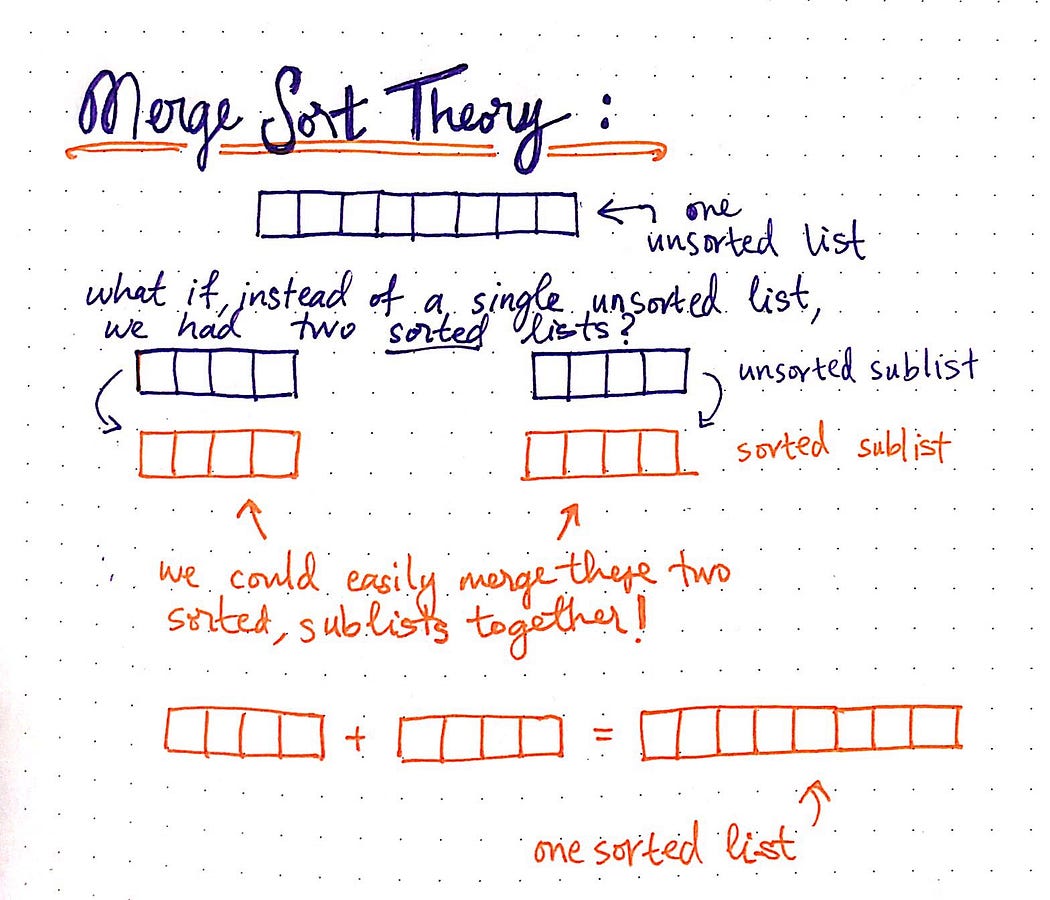
|  |
| --- |
| The merge sort algorithm is a sorting algorithm that sorts a collection by breaking it into half. It then sorts those two halves, and then merges them together, in order to form one, completely sorted collection. |



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 t**hrough 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.

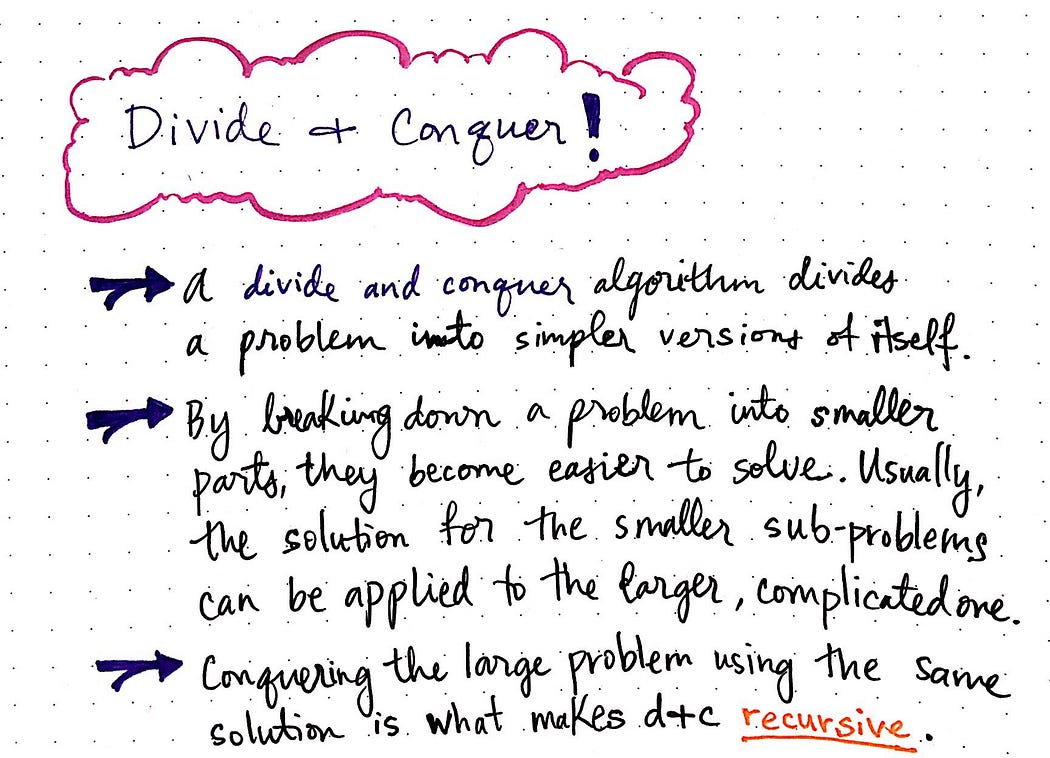
This is where merge sort fundamentally differs from all of these other sorting algorithms. Let’s take a look at an example to help illustrate this.



* 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:**

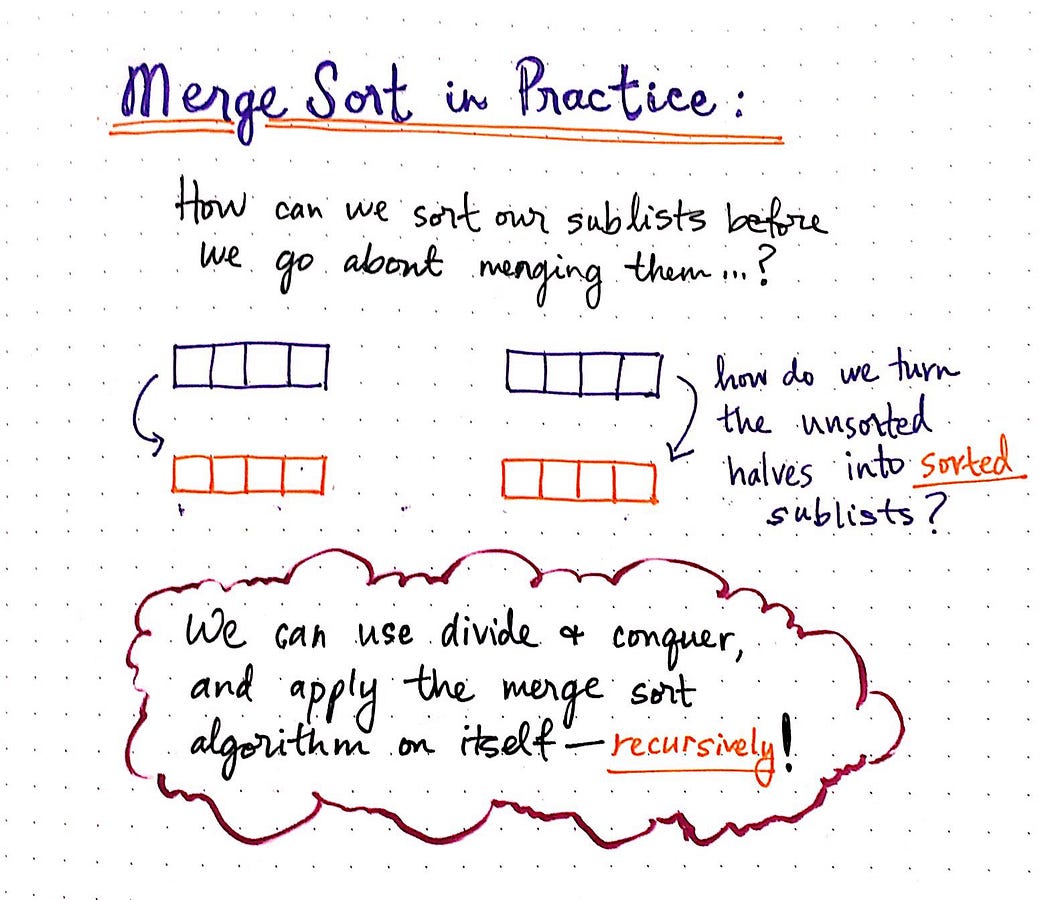
* Divide and break up the problem into the smallest possible **“subproblem”, of the exact same type.**
* Conquer and tackle the smallest subproblems first. Once you’ve figured out a solution that works, use that exact same technique to solve the larger subproblems — in other words, solve the subproblems recursively.
* 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!



* 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**

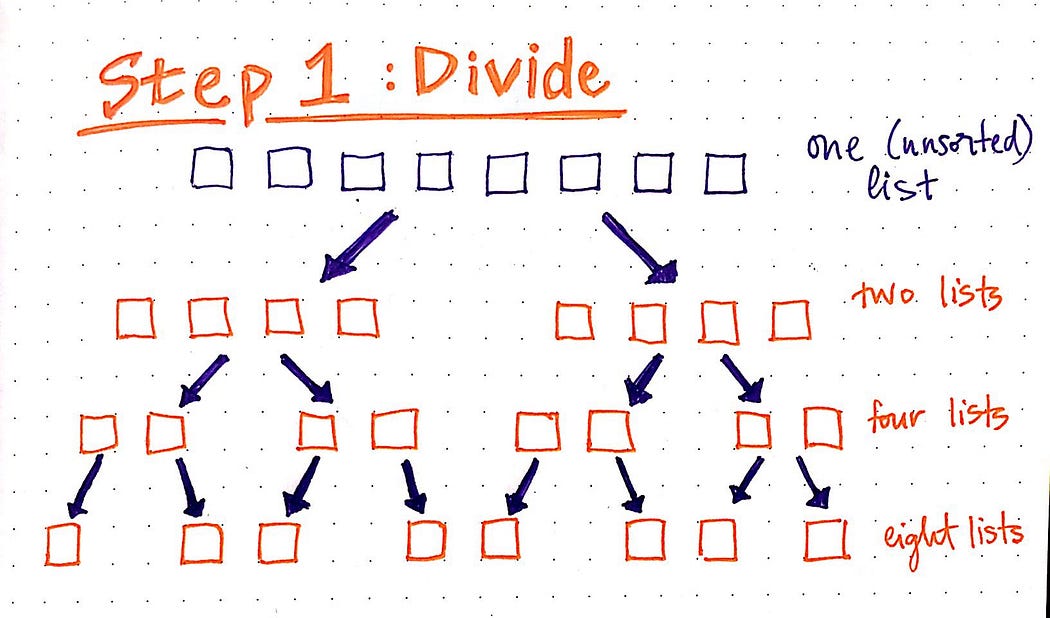
* Understanding divide and conquer in theory is one thing, but its usefulness tends to be far more obvious if we can see it in action.
* Let’s take a look at how we can use recursion to divide and conquer the illusive merge sort algorithm!



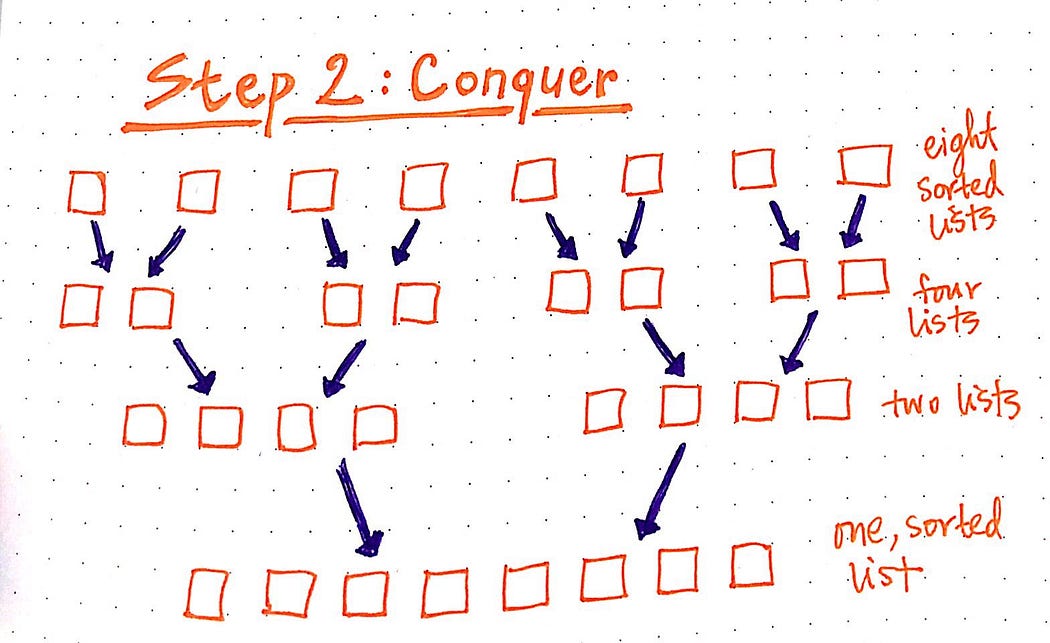
In theory, we understand that merge sort splits up an unsorted list, sorts the two halves, and merges them together.

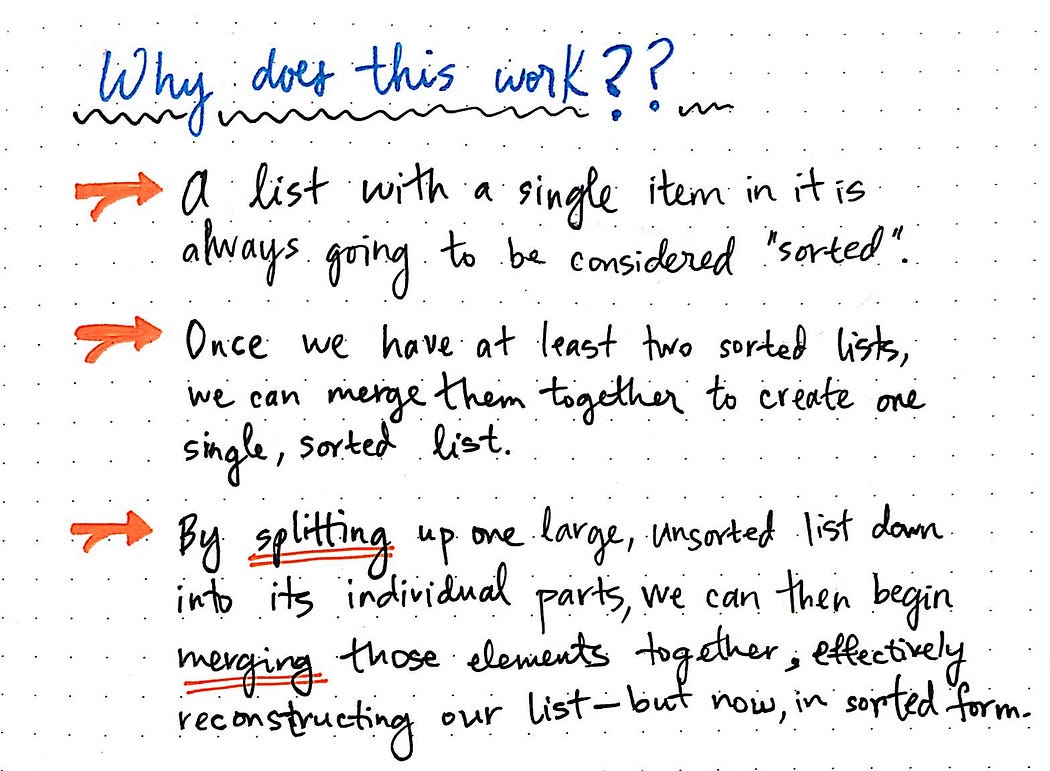
**But how does this algorithm really work in practice?**

* 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.

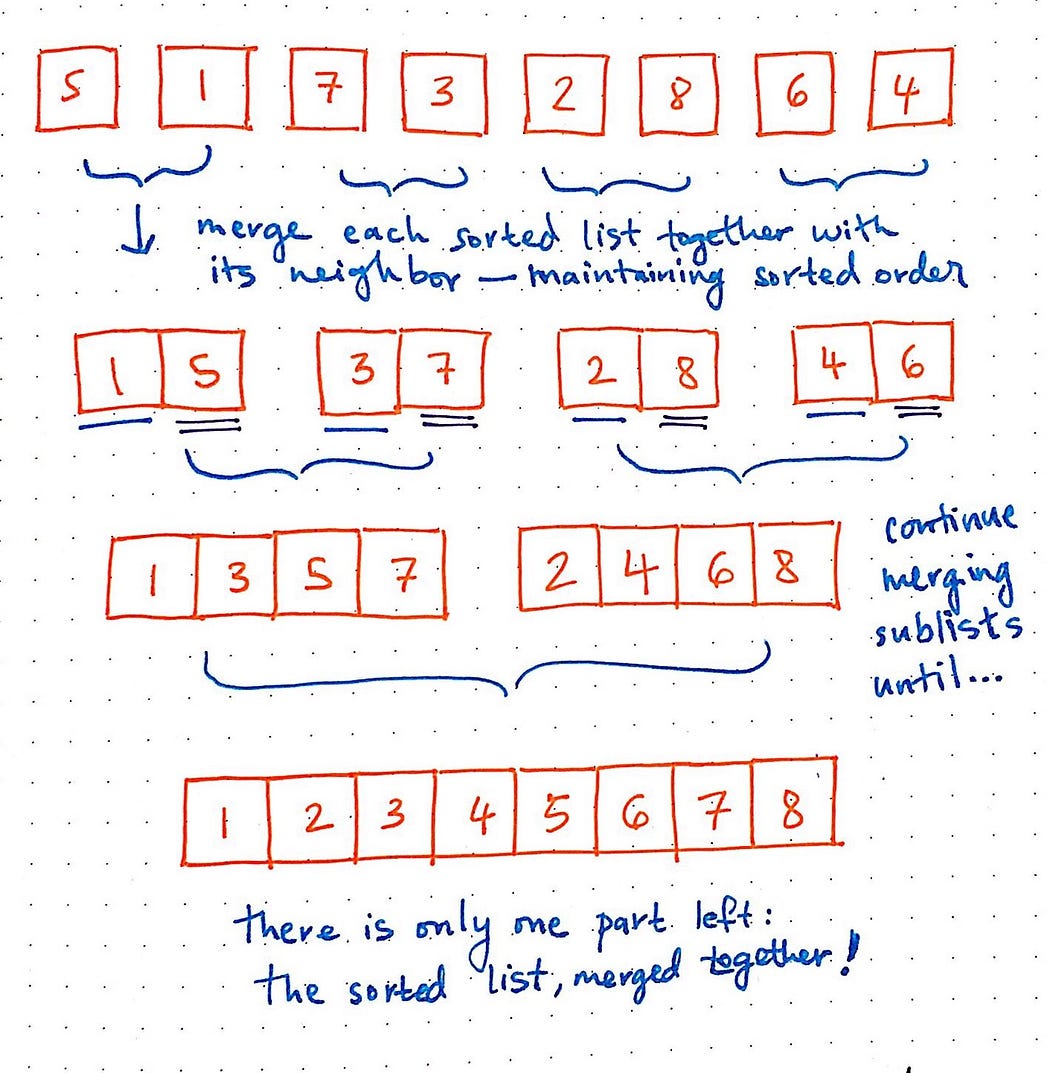




This is exactly what makes recursion so powerful: once we’ve figured out how to solve the problem of merging two lists together, it doesn’t matter if the list has one element, or a hundred — we can reuse the same logic in both scenarios.

***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 continuing dividing until we hit our base case.
* 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.



* 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**

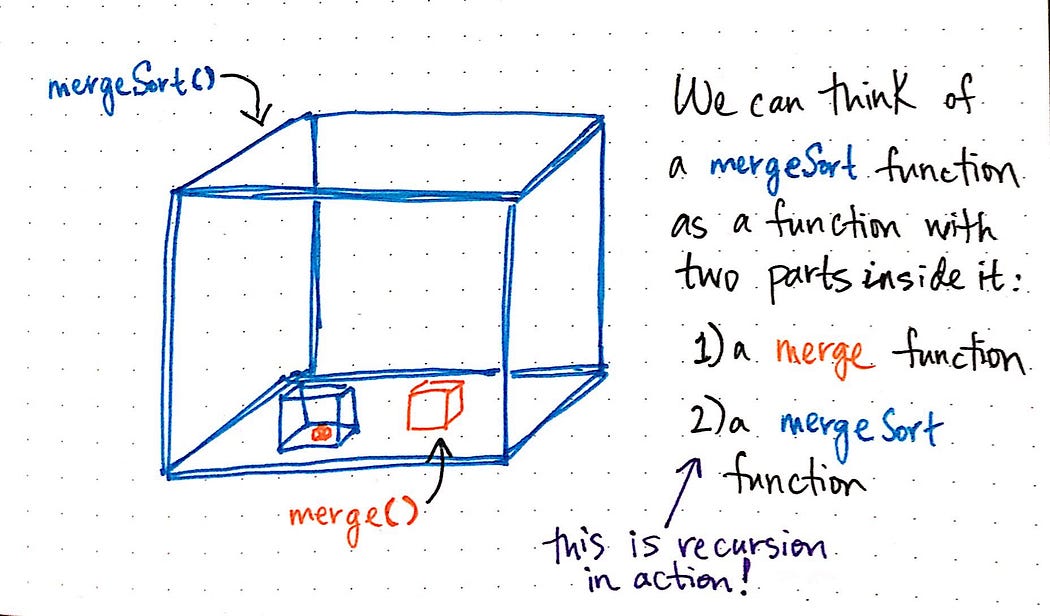
* We already know that a recursive algorithm is one that conceptually uses the same solution to solve a problem. When it comes to code, this can effectively be translated to a function that calls itself.
* In the code below — which is adapted from Rosetta Code’s JavaScript implementation of merge sort — we can see that the mergeSort function actually calls itself.

**Program:**

|  |
| --- |
| package com.arrays.utility.methods.sort;  public class MergeSortDemo {  public static void main(String[] args) {    int[] arr = {7,1,5,2,3,9};    mergeSort(arr, arr.length);    for(int i : arr) {  System.out.println(i);  }    }    public static void mergeSort(int[] arr, int n) {    if(n == 1) {  return;  }    int mid = n/2;  int[] l = new int[mid];  int[] r = new int[n-mid];    for(int i=0; i<mid; i++) {  l[i] = arr[i];  }  for(int j= mid; j<n;j++) {  r[j - mid] = arr[j];  }    mergeSort(l, mid);  mergeSort(r, n-mid);    merge(arr, l, r, mid, n-mid);  }    public static void merge(int[] arr, int[] leftArray, int[] rightArray, int leftArrayEndIndex, int rightArrayEndIndex) {    int i = 0,j = 0,k = 0;    while(i< leftArrayEndIndex && j < rightArrayEndIndex) {  if(leftArray[i] <= rightArray[j]) {  arr[k++] = leftArray[i++];  } else {  arr[k++] = rightArray[j++];  }  }    while(i < leftArrayEndIndex) {  arr[k++] = leftArray[i++];  }    while(j < rightArrayEndIndex) {  arr[k++] = rightArray[j++];  }    }    } |

**Explanation:**

* 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!



**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.

**Merge Sort in Java Script:**

|  |
| --- |
| function mergeSort(array) {  // Determine the size of the input array.  var arraySize = array.length;    // If the array being passed in has only one element  // within it, it is considered to be a sorted array.  if (arraySize === 1) {  return;  }    // If array contains more than one element,  // split it into two parts (left and right arrays).  var midpoint = Math.floor(arraySize / 2);  var leftArray = array.slice(0, midpoint);  var rightArray = array.slice(midpoint);    // Recursively call mergeSort() on  // leftArray and rightArray sublists.  mergeSort(leftArray);  mergeSort(rightArray);    // After the mergeSort functions above finish executing,  // merge the sorted leftArray and rightArray together.  merge(leftArray, rightArray, array);    // Return the fully sorted array.  return array;  }  function merge(leftArray, rightArray, array) {  var index = 0;    while (leftArray.length && rightArray.length) {  console.log('array is: ', array);  if (rightArray[0] < leftArray[0]) {  array[index++] = rightArray.shift();  } else {  array[index++] = leftArray.shift();  }  }    while (leftArray.length) {  console.log('left array is: ', leftArray);  array[index++] = leftArray.shift();  }    while (rightArray.length) {  console.log('right array is: ', rightArray);  array[index++] = rightArray.shift();  }    console.log('\*\* end of merge function \*\* array is: ', array);  } |

What happens when we run this code? Well, let’s try sorting an array of of eight elements: **[5, 1, 7, 3, 2, 8, 6, 4].** I’ve added some console.log's to make it a little bit easier to see how this algorithm works under the hood.

|  |
| --- |
| var array = [5, 1, 7, 3, 2, 8, 6, 4];  mergeSort(array);  > array is: (2) [5, 1]  > left array is: [5]  > \*\* end of merge function \*\* array is: (2) [1, 5]  > array is: (2) [7, 3]  > left array is: [7]  > \*\* end of merge function \*\* array is: (2) [3, 7]  > array is: (4) [5, 1, 7, 3]  > array is: (4) [1, 1, 7, 3]  > array is: (4) [1, 3, 7, 3]  > right array is: [7]  > \*\* end of merge function \*\* array is: (4) [1, 3, 5, 7]  > array is: (2) [2, 8]  > right array is: [8]  > \*\* end of merge function \*\* array is: (2) [2, 8]  > array is: (2) [6, 4]  > left array is: [6]  > \*\* end of merge function \*\* array is: (2) [4, 6]  > array is: (4) [2, 8, 6, 4]  > array is: (4) [2, 8, 6, 4]  > array is: (4) [2, 4, 6, 4]  > left array is: [8]  > \*\* end of merge function \*\* array is: (4) [2, 4, 6, 8]  > array is: (8) [5, 1, 7, 3, 2, 8, 6, 4]  > array is: (8) [1, 1, 7, 3, 2, 8, 6, 4]  > array is: (8) [1, 2, 7, 3, 2, 8, 6, 4]  > array is: (8) [1, 2, 3, 3, 2, 8, 6, 4]  > array is: (8) [1, 2, 3, 4, 2, 8, 6, 4]  > array is: (8) [1, 2, 3, 4, 5, 8, 6, 4]  > array is: (8) [1, 2, 3, 4, 5, 6, 6, 4]  > right array is: [8]  > \*\* end of merge function \*\* array is: (8) [1, 2, 3, 4, 5, 6, 7, 8]  >> (8) [1, 2, 3, 4, 5, 6, 7, 8]> |

**Explanation:**

* Rad! Look at all of that merging going on! Because of all the things that we’re logging out, we can see how this algorithm recursively just keeps on dividing elements until it has single-item lists: for example, it starts off with [5] and [1].
* We can also see that, as we merge together two sublists and build up our array, we’re also doing the work of sorting the elements and putting them in their correct order.
* Notice how this happens with the two sublists of [1, 5] and [7, 3].
* This algorithm uses a temporary structure (usually an array), and adds the sublists to that temporary array, in sorted order.
* Only after the sublists have been sorted and added will it actually proceed to copy those “sorted sublists” into the original array.

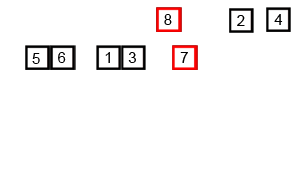
**But back to our original mission:** merge sort’s speed and efficiency.

* In order for us to understand that, we need to look at how must time it takes for the merge sort algorithm to do the work of

1) dividing the collection, 2) sorting it, and 3) merging it all back together.

* Well, neither dividing the collection nor sorting it is the worst part of this algorithm.
* Because we’re using recursion — and because we’re splitting each half into halves recursively — **the work of dividing the collection isn’t too expensive**.

Furthermore, since we sort two items at a time using a comparison operator (**like < or >),** we know that this, too can’t be all that expensive **— it should take a constant amount of time.**

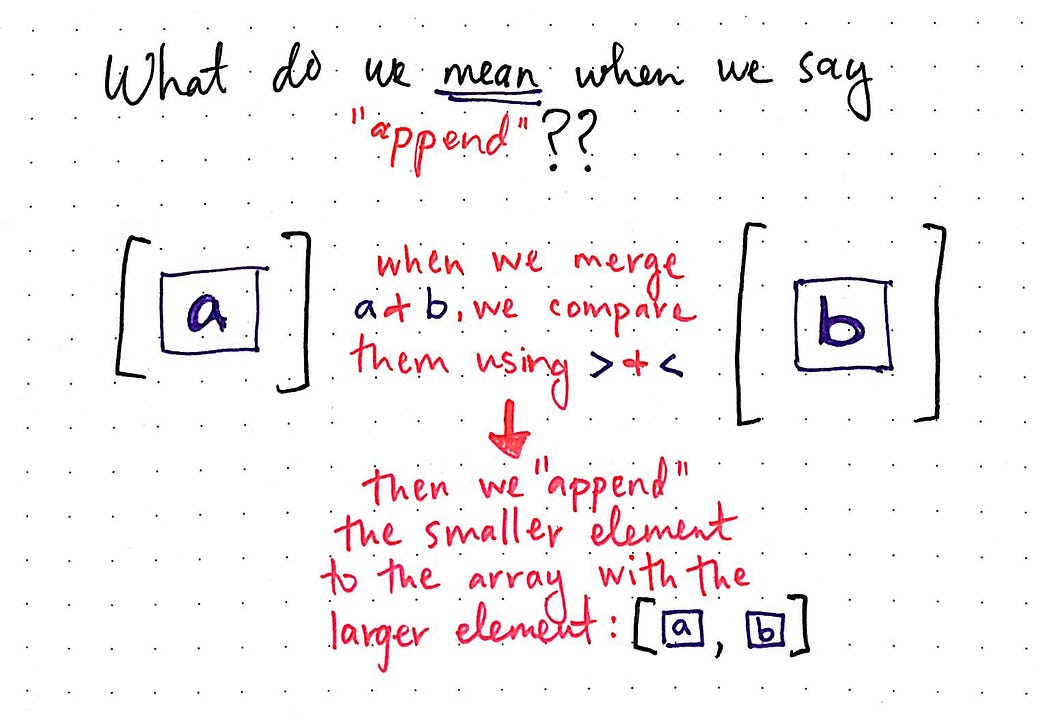


* **Instead, the most expensive part of any merge sort algorithm is the work of merging, itself** — the work of putting back together the small sublists, in sorted format.
* If we go back and look at our code again, we can see that the work of merging is actually taking up the most amount of time, and what is logged out the most often, too.
* Thus, we’ll focus on the aspect of merging, since that’s what is responsible for giving the algorithm it’s time complexity. So, what do we mean when we say ‘merging’?

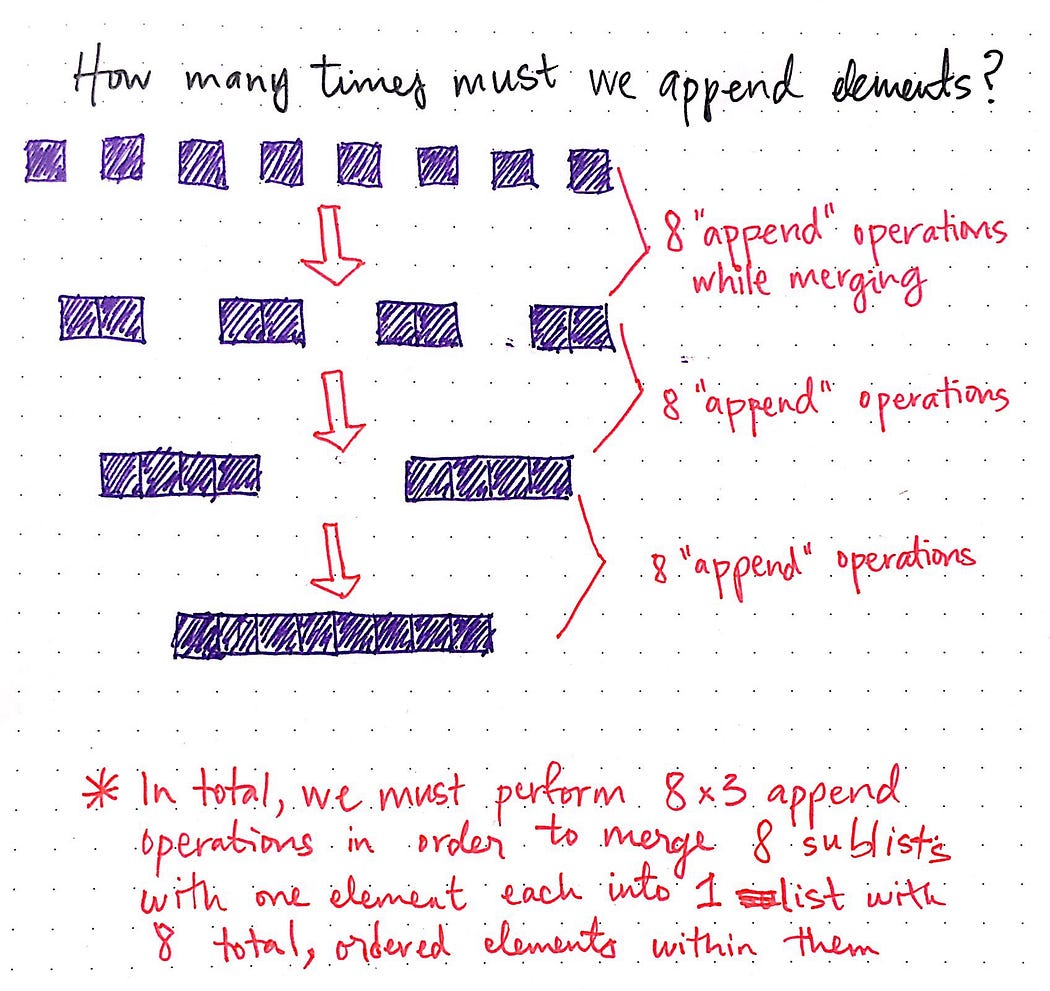
|  |
| --- |
| When we use the term ‘merging’, what we’re actually referring to is the act of appending an item — in sorted order — to the temporary array structure, which we build up over time as we continue to sort and append more items to it. |

* After the merge sort algorithm has recursively divide up the unsorted list into single-item elements, we can say that, since each list has only one item in it, everything is sorted (but obviously not joined together yet).
* The step of merging involves comparing two sublists, and then appending the larger one to the end of the smaller one.

This will probably make a bit more sense with an example:



* In the example above, we can imagine that we’re sorting a large dataset, and we’re at the point now where we’ve divided the collection down to its recursive base case: the point where each list has only one element in it.
* Here, we have two sublists, one with the element a and another with the element b. In order to merge these two items together, we must determine which of the **two is smaller (using a comparison operator),** and then append one element to the other.
* If we look at different merge sort algorithm implementations, we’ll probably notice pretty quickly that people will often write their merge functions in slightly different ways, **depending on whether they’re using the greater than (>) or less than (<) operator.** Regardless of individual implementations, however, the basic idea of how we do the work of merging tends to remain the same.
* Okay — in this particular example, we had two lists, each with only one element in it. We had to perform a single append operation in order to merge them together. Realistically however, **we’re never really going to be sorting a list of two items, right?** Instead, we’ll be dealing with a much larger dataset. Let’s see how many times we’d have to append elements with a bigger unsorted collection. In the example shown below, instead of two items, we have a collection of eight items that need to be sorted.



We’ve already done the work of dividing, and now we’re ready to start merging things together. We begin with eight, individual items in eight lists — remember that they’re considered “sorted” because there’s only one item in them! We want to combine them together such that we have half as many lists.

If we think about it, this makes sense: when we were dividing, we were doubling the number of lists until we reached our base case of one item per list. Now, we’re doing the exact opposite in order to build our list up.

To start, if we have eight items, we need to merge them together in sorted order such that we have just four items. This means we’ll need to perform eight “append” operations.

**Remember that an “append” operation involves two steps:**

1. comparing the items that we want to combine together, and
2. inserting them into our temporary array in sorted order

Effectively, we’ll need to perform one append operation per item. Once we’ve done that, we’ll end up with four lists, with each list’s elements in sorted order. Great!

Next step: we’ll need to merge our four lists together so that we have just two lists. We’ll again need to perform eight append operations in the process of comparing, sorting, and inserting the elements into their correct places.

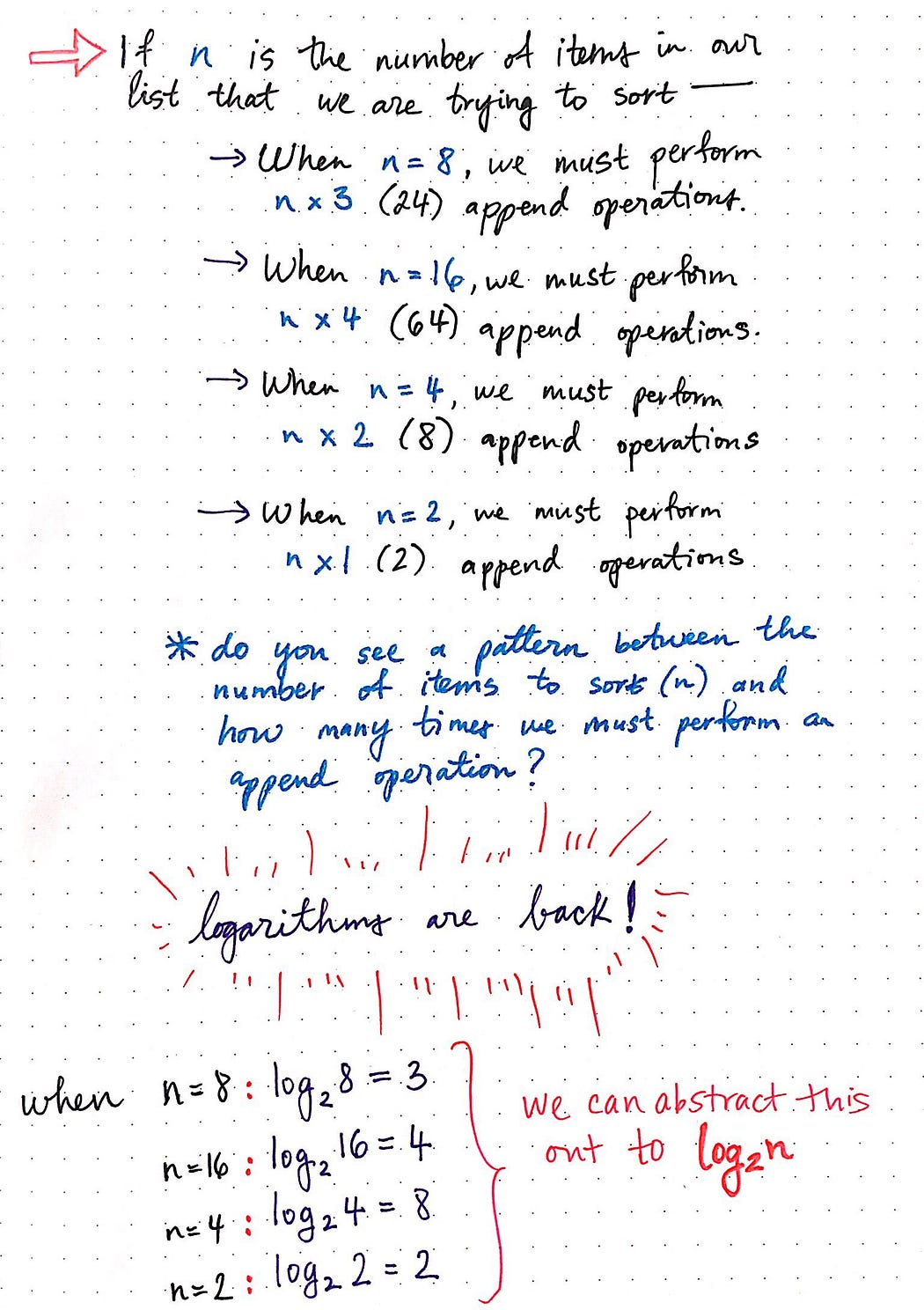
Alright, now we’re down to just two lists, with four sorted elements in each of them! You know what’s coming, right? We’ll perform eight append operations, and end up with one single, sorted list that has eight elements within it.

Great! We’re done sorting — finally! Let’s take a step back and look at what just happened. In total, we **performed (8 + 8 + 8), or 24** append operations in order to turn out single-element sublists into one, sorted list. In other words, we performed 8 × 3 append operations. I’m repeating this a few different times, because there’s a special relationship between those numbers, which you might remember from an earlier post.

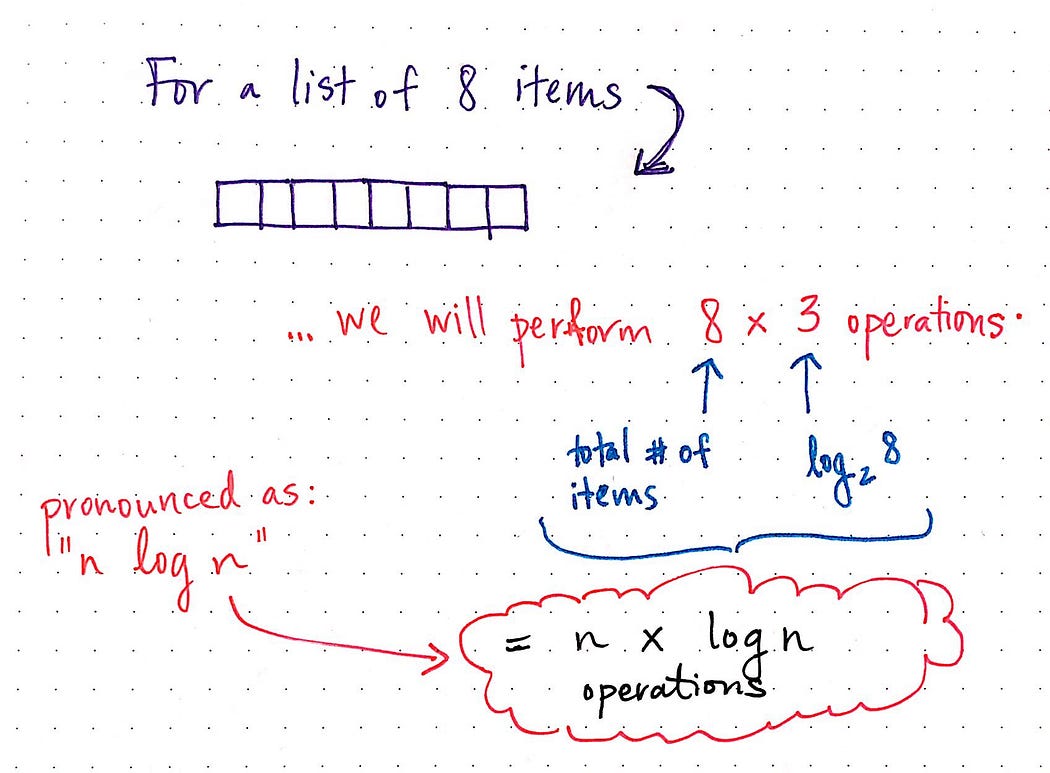
If not, don’t worry! We’ll look at some more examples and see if we can identify the relationship and any patterns.

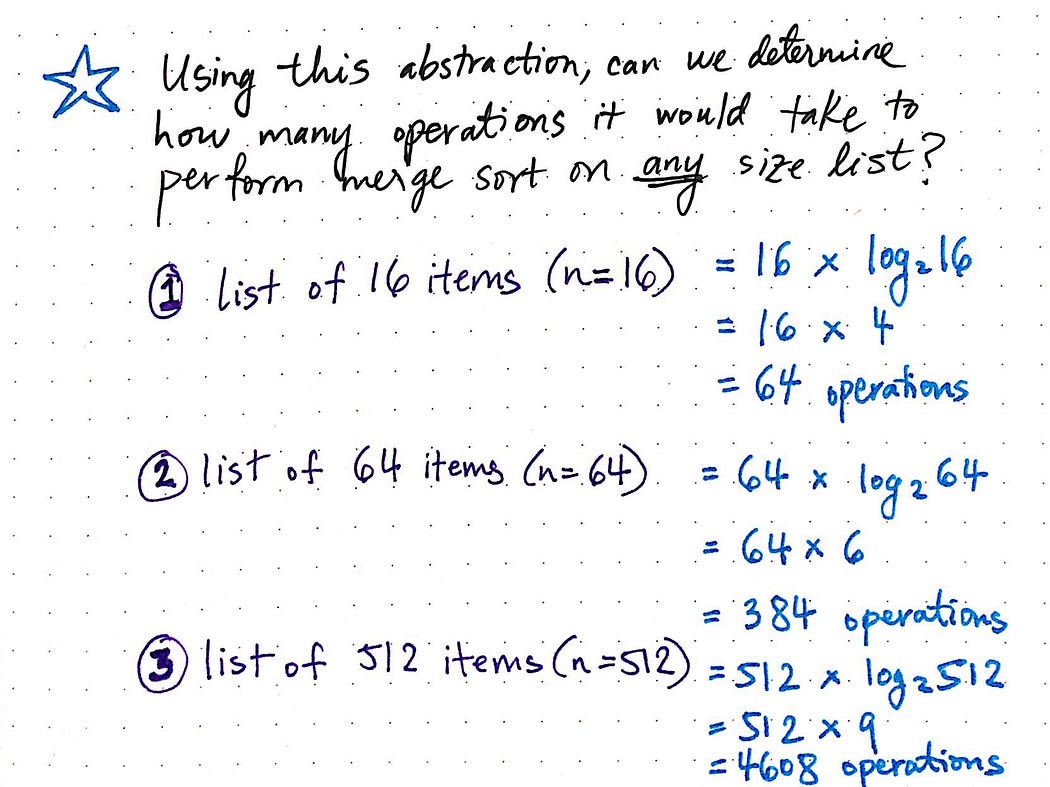
**Logarithms strike back — but like, in a nice way**

One of my favorite authors once wrote that “Brevity is the soul of wit”, and I tend to believe him. So, rather than illustrate a ton of long examples, I’ve just listed out how many append operations they’ll require.

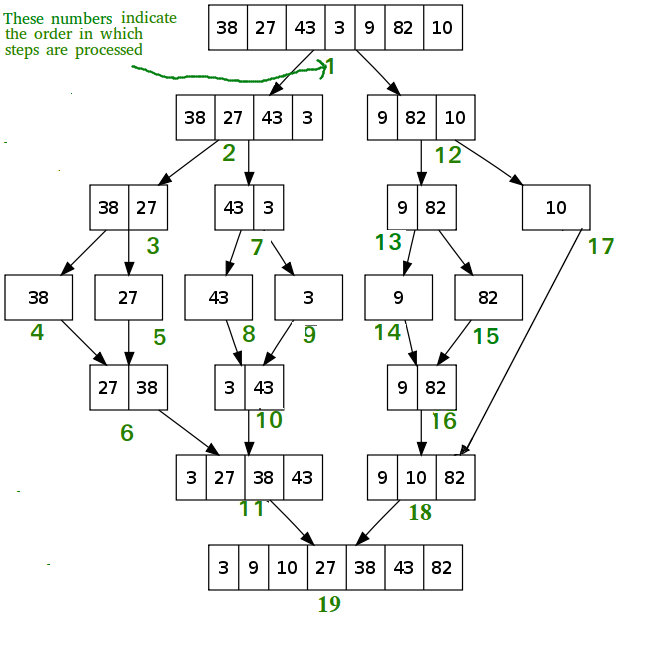


* We already know that, when the number of elements in the list, which we’ll call n, is 8, our merge sort algorithm will perform 8 × 3 append operations.
* When n = 16, we must perform 16 × 4 append operations. When n = 4, merge sort will perform 4 × 2 append operations. And when n = 2, merge sort will perform 2 × 1 append operations.
* If we abstract out n in each of these scenarios, we’ll start to see something familiar emerge.
* Tada! Logarithms are back. If you’re unfamiliar with how logs work, check out this post from earlier in the year. I promise that it’ll give tons more context on what’s happening here.
* If we abstract out n, we’ll see that in all of these examples, one truth seems to be evident:
* If we multiply the log of n by the value of n, the result ends up being the number of total append operations to perform.
* So what does that mean, exactly? I think that this statement can be a little bit confusing, but it should (hopefully) make more sense if we visualize it. Let’s return to our original example of performing merge sort on a list of eight elements:





**Time and Space Complexity Analysis of Merge Sort:**



The Time Complexity of merge sort for Best case, average case and worst case is O(N \* logN).

T(k) = time taken to sort k elements

M(k) = time taken to merge k elements

So, it can be written

T(N) = 2 \* T(N/2) + M(N)

= 2 \* T(N/2) + constant \* N

These N/2 elements are further divided into two halves. So,

T(N) = 2 \* [2 \* T(N/4) + constant \* N/2] + constant \* N

= 4 \* T(N/4) + 2 \* N \* constant. . .

= 2k \* T(N/2k) + k \* N \* constant

It can be divided maximum until there is one element left. So, then N/2k = 1. k = log2N

T(N) = N \* T(1) + N \* log2N \* constant

= N + N \* log2N

**Therefore the time complexity is O(N \* log2N).**

So in the best case, the worst case and the average case the time complexity is the same.

**Space Complexity Analysis:**

In case of merge sort, we need to build a **temporary array** to merge the sorted arrays. So the auxiliary space requirement is O(N).

Feeling lost in the world of random DSA topics, wasting time without progress? It's time for a change! Join our DSA course, where we'll guide you on an exciting journey to master DSA efficiently and on schedule.

**Bubble Sort Algorithm:**

Bubble Sort is a basic comparison-based sorting algorithm that works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The process is named “bubble sort” because the smaller elements “bubble” to the top of the list while the larger elements “sink” to the bottom.

Despite its simplicity, Bubble Sort is not the most efficient sorting algorithm, especially for large datasets. Its average and worst-case time complexity is O(n²), where ’n’ is the number of elements to be sorted. However, it provides a clear introduction to the concept of sorting algorithms.

**How Bubble Sort Works: Step-by-Step Explanation**

Let’s walk through the Bubble Sort process step by step:

1. Start by comparing the first two elements of the list.
2. If the first element is greater than the second element, swap them.
3. Move to the next pair of elements (the second and third elements), and repeat the comparison and swapping process.
4. Continue this process until you reach the end of the list. By this point, the largest element will have “**bubbled up”** to the **end of the list.**
5. Repeat steps 1–4 for the remaining elements, excluding the last element since it is already in its correct position.
6. Repeat the entire process for a decreasing range of elements until the entire list is sorted.

Implementing Bubble Sort in Java: Code Example

Here’s a Java implementation of the Bubble Sort algorithm: