Overview of Merge Sort

Because we're using divide-and-conquer to sort, we need to decide what our subproblems are going to look like. The full problem is to sort an entire array. Let's say that a subproblem is to sort a subarray. In particular, we'll think of a subproblem as sorting the subarray starting at index pp and going through index rr. It will be convenient to have a notation for a subarray, so let's say that array[p..r] denotes this subarray of array. Note that this "two-dot" notation is not legal; we're using it just to describe the algorithm, rather than a particular implementation of the algorithm in code. In terms of our notation, for an array of **n** elements, we can say that the original problem is to sort **array[0..n-1]**.

Here's how merge sort uses divide-and-conquer:

1. Divide by finding the number **q** of the position midway between **p** and **r**. Do this step the same way we found the midpoint in binary search: add **p** and **r**, divide by 2, and round down.
2. Conquer by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray **array[p..q]** and recursively sort the subarray **array[q+1..r]**.
3. Combine by merging the two sorted subarrays back into the single sorted subarray **array[p..r]**.

We need a base case. The base case is a subarray containing fewer than two elements, that is, when $p \geq r$, since a subarray with no elements or just one element is already sorted. So we'll divide-conquer-combine only when $p < r$.

Let's see an example. Let's start with array holding [14, 7, 3, 12, 9, 11, 6, 2], so that the first subarray is actually the full array, **array[0..7] (p=0 and r=7)**. This subarray has at least two elements, and so it's not a base case.

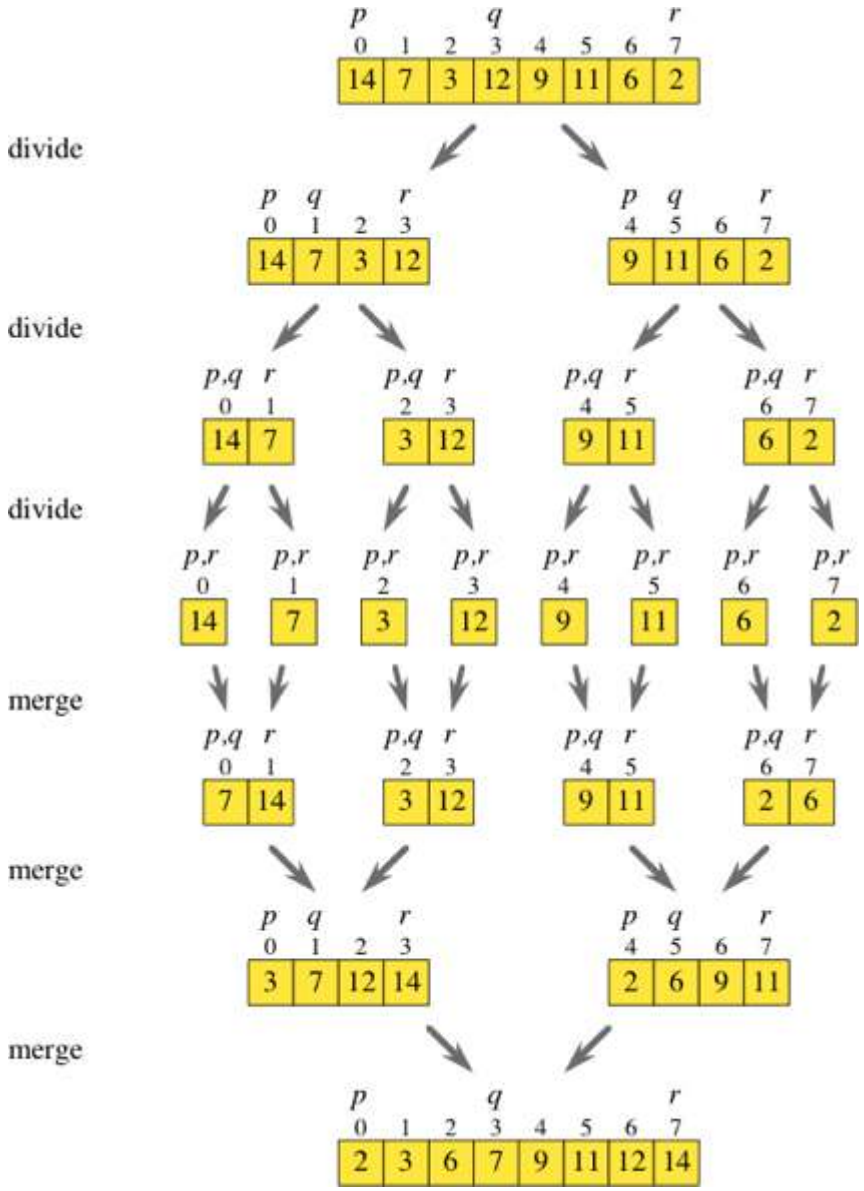
subarray has at least two elements, and so it's not a base case.

- In the divide step, we compute **q=3**.
- The conquer step has us sort the two subarrays **array[0..3]**, which contains [14, 7, 3, 12], and **array[4..7]**, which contains [9, 11, 6, 2]. When we come back from the conquer step, each of the two subarrays is sorted: array[0..3] contains [3, 7, 12, 14] and array[4..7] contains [2, 6, 9, 11], so that the full array is [3, 7, 12, 14, 2, 6, 9, 11].
- Finally, the **combine** step merges the two sorted subarrays in the first half and the second half, producing the final sorted array [2, 3, 6, 7, 9, 11, 12, 14].

How did the subarray **array[0..3]** become sorted? The same way. It has more than two elements, and so it's not a base case. With **p=0**and **r=3**, compute **q=1**, recursively sort **array[0..1] ([14, 7])** and **array[2..3] ([3, 12])**, resulting in **array[0..3]** containing [7, 14, 3, 12], and merge the first half with the second half, producing [3, 7, 12, 14].

How did the subarray **array[0..1]** become sorted? With **p=0** and **r=1**, compute **q=0**, recursively sort **array[0..0]** ([14]) and **array[1..1]** ([7]), resulting in **array[0..1]** still containing [14, 7], and merge the first half with the second half, producing [7, 14].

The subarrays **array[0..0]** and **array[1..1]** are base cases, since each contains fewer than two elements. Here is how the entire merge sort algorithm unfolds:



Most of the steps in merge sort are simple. You can check for the base case easily. Finding the midpoint qq in the divide step is also really easy. You have to make two recursive calls in the conquer step. It's the combine step, where you have to merge two sorted subarrays, where the real work happens. For the moment, let's assume that we know how to merge two sorted subarrays efficiently and continue to focus on merge sort as a whole.