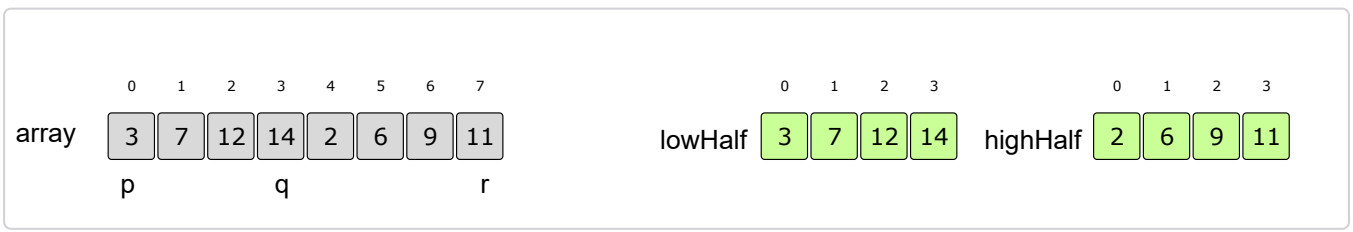
Linear-time Merging

The remaining piece of merge sort is the merge function, which merges two adjacent sorted subarrays, **array[p..q]** and **array[q+1..r]** into a single sorted subarray in **array[p..r]**. We'll see how to construct this function so that it's as efficient as possible. Let's say that the two subarrays have a total of **n** elements. We have to examine each of the elements in order to merge them together, and so the best we can hope for would be a merging time of **Θ(n)**. Indeed, we'll see how to merge a total of **n** elements in **Θ(n)** time.

In order to merge the sorted subarrays **array[p..q]** and **array[q+1..r]** and have the result in **array[p..r]**, we first need to make temporary arrays and copy **array[p..q]** and **array[q+1..r]** into these temporary arrays. We can't write over the positions in **array[p..r]** until we have the elements originally in **array[p..q]** and **array[q+1..r]** safely copied.

The first order of business in the merge function, therefore, is to allocate two temporary arrays, **lowHalf** and **highHalf**, to copy all the elements in **array[p..q]** into **lowHalf**, and to copy all the elements in **array[q+1..r]** into **highHalf**. How big should **lowHalf** be? The subarray **array[p..q]** contains **q−p+1** elements. How about **highHalf**? The subarray **array[q+1..r]** contains **r−q** elements. (In JavaScript, we don't have to give the size of an array when we create it, but since we do have to do that in many other programming languages, we often consider it when describing an algorithm.)

In our example array [14, 7, 3, 12, 9, 11, 6, 2], here's what things look like after we've recursively sorted **array[0..3]** and **array[4..7]** (so that **p=0**, **q=3**, and **r=7**) and copied these subarrays into **lowHalf** and **highHalf**:

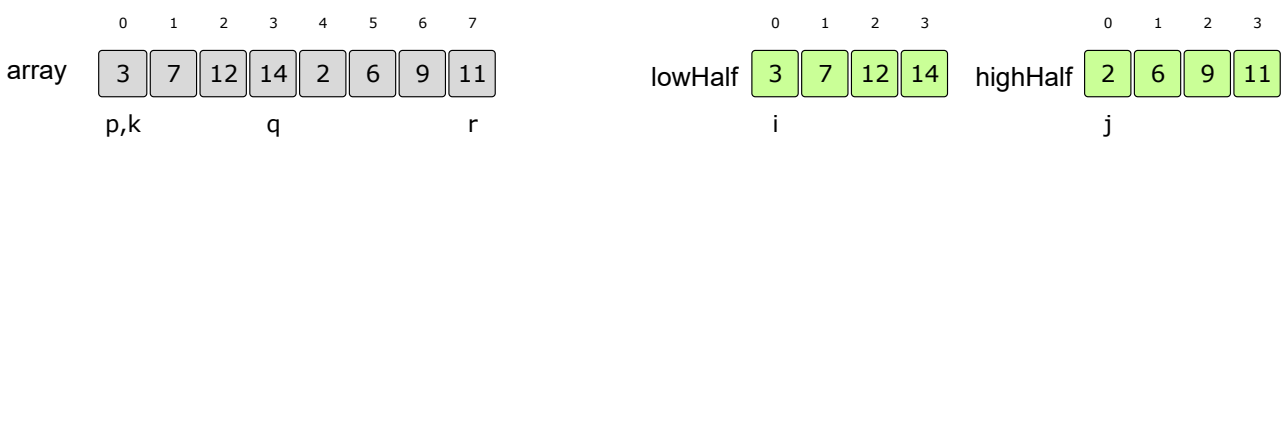


The numbers in array are grayed out to indicate that although these array positions contain values, the "real" values are now in **lowHalf** and **highHalf**. We may overwrite the grayed numbers at will.

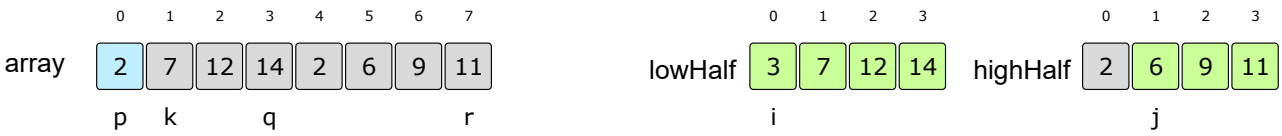
Next, we merge the two sorted subarrays, now in **lowHalf** and **highHalf**, back into **array[p..r]**. We should put the smallest value in either of the two subarrays into **array[p]**. Where might this smallest value reside? Because the subarrays are sorted, the smallest value must be in one of just two places: either **lowHalf[0]** or **highHalf[0]**. (It's possible that the same value is in both places, and then we can call either one the smallest value.) With just one comparison, we can determine whether to copy **lowHalf[0]** or **highHalf[0]** into **array[p]**. In our example, **highHalf[0]** was smaller. Let's also establish three variables to index into the arrays:

- i indexes the next element of **lowHalf** that we have not copied back into array. Initially, i is 0.
- j indexes the next element of **highHalf** that we have not copied back into array. Initially, j is 0.
- k indexes the next location in array that we copy into. Initially, k equals p.

After we copy from **lowHalf** or **highHalf** into array, we must increment (add 1 to) k so that we copy the next smallest element into the next position of array. We also have to increment i if we copied from lowHalf, or increment j if we copied from highHalf. So here are the arrays before and after the first element is copied back into array:

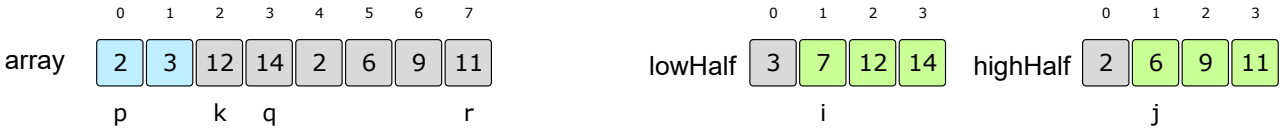


1 of 9



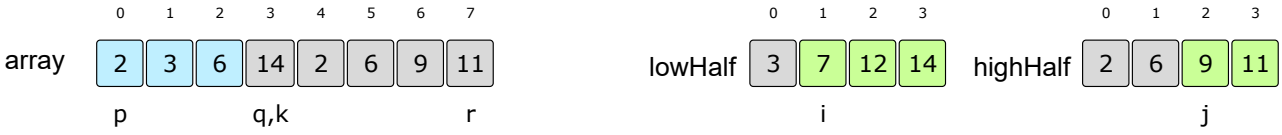
We've grayed out highHalf[0] to indicate that it no longer contains a value that we're going to consider.
The unmerged part of the highHalf array starts at index j, which is now 1.
The value in array[p] is no longer grayed out, because we copied a "real" value into it.

2 of 9



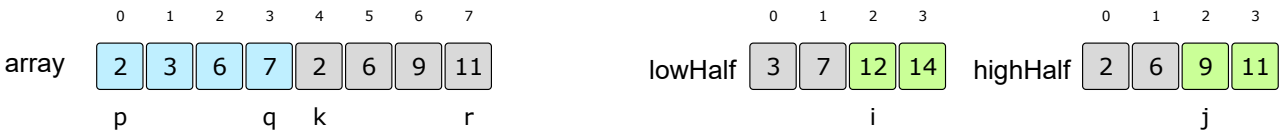
Where must the next value to copy back into array reside?
It's either first untaken element in lowHalf (lowHalf[0]) or the first untaken element in highHalf (highHalf[1])
With one comparison, we determine that lowHalf[0] is smaller, and so we copy it into array[k] and increment k and i

3 of 9

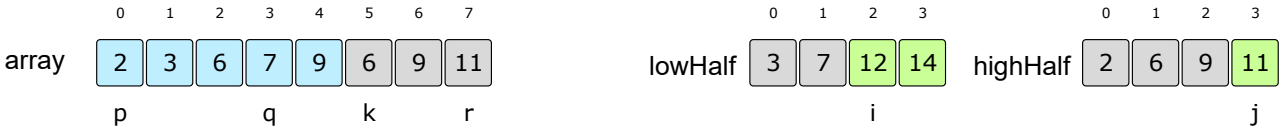


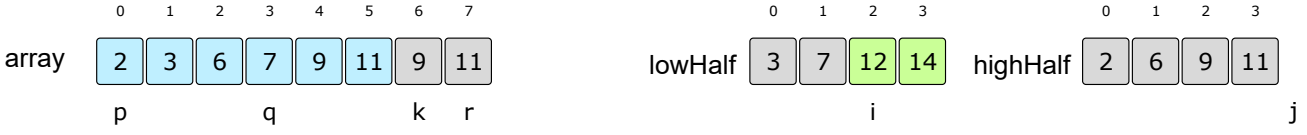
Next, we compare lowHalf[1] and highHalf[1], determining that we should copy highHalf[1] into array[k].
We then increment k and j

4 of 9

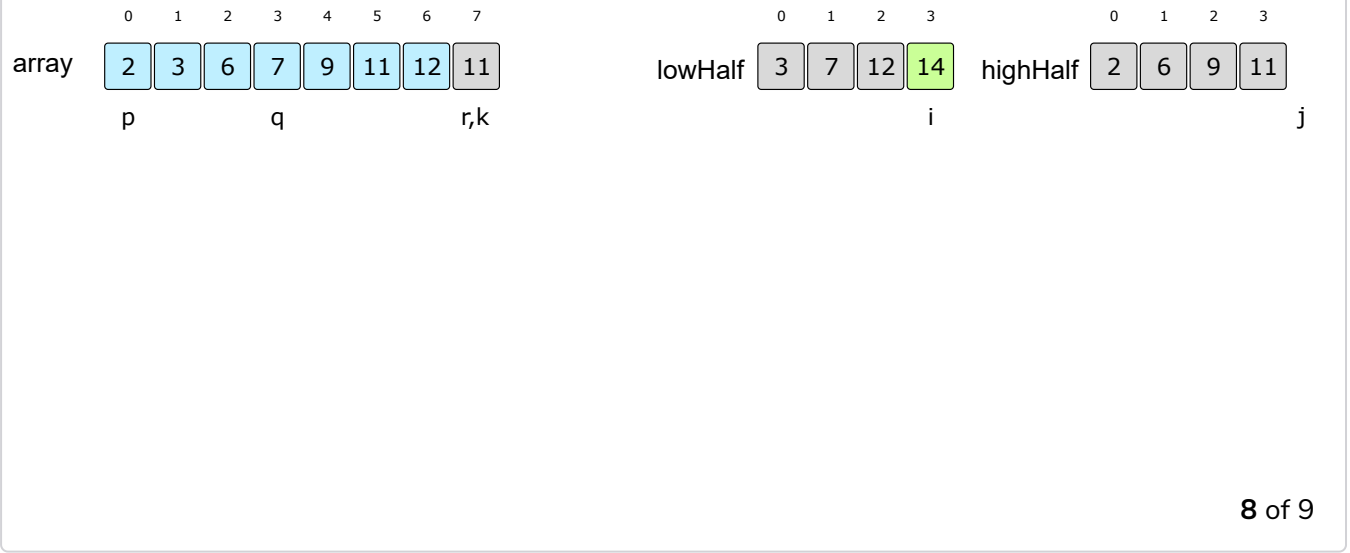


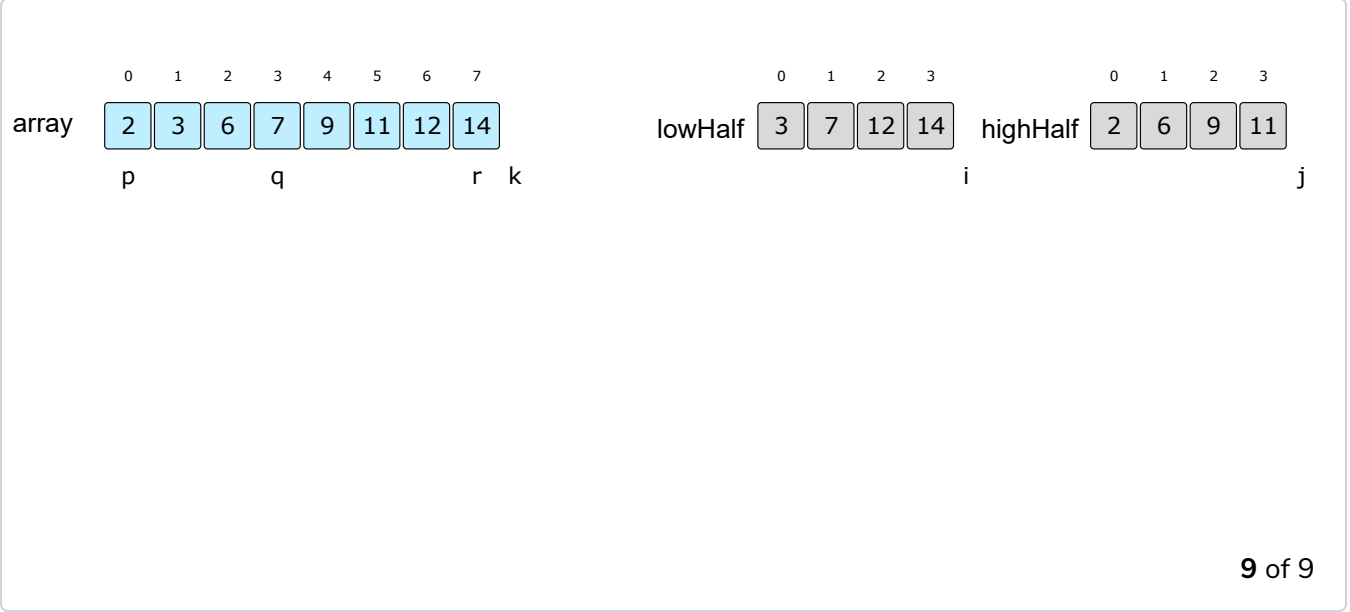
Keep going, always comparing lowHalf[i] and highHalf[j], copying the smaller of the two into array[k], and incrementing either i or j





Eventually, either all of lowHalf or all of highHalf is copied back into array. In this example, all of highHalf is copied back before the last few elements of lowHalf. We finish up by just copying the remaining untaken elements in either lowHalf or highHalf:





We claimed that merging **n** elements takes **Θ(n)** time, and therefore the running time of merging is linear in the subarray size. Let's see why this is true. We saw three parts to merging:

1. Copy each element in **array[p..r]** into either **lowHalf** or **highHalf**.
2. As long as some elements are untaken in both **lowHalf** and **highHalf**, compare the first two untaken elements and copy the smaller one back into array.
3. Once one of **lowHalf** and **highHalf** has had all its elements copied back into array, copy each remaining untaken element from the other temporary array back into array.

How many lines of code do we need to execute for each of these steps? It's a constant number per element. Each element is copied from array into either lowHalf or highHalf exactly one time in step 1. Each comparison in step 2 takes constant time, since it compares just two elements, and each element "wins" a comparison at most one time. Each element is copied back into array exactly one time in steps 2 and 3 combined. Since we execute each line of code a constant number of times per element and we assume that the subarray array[p..q] contains **n** elements, the running time for merging is indeed **Θ(n)**.