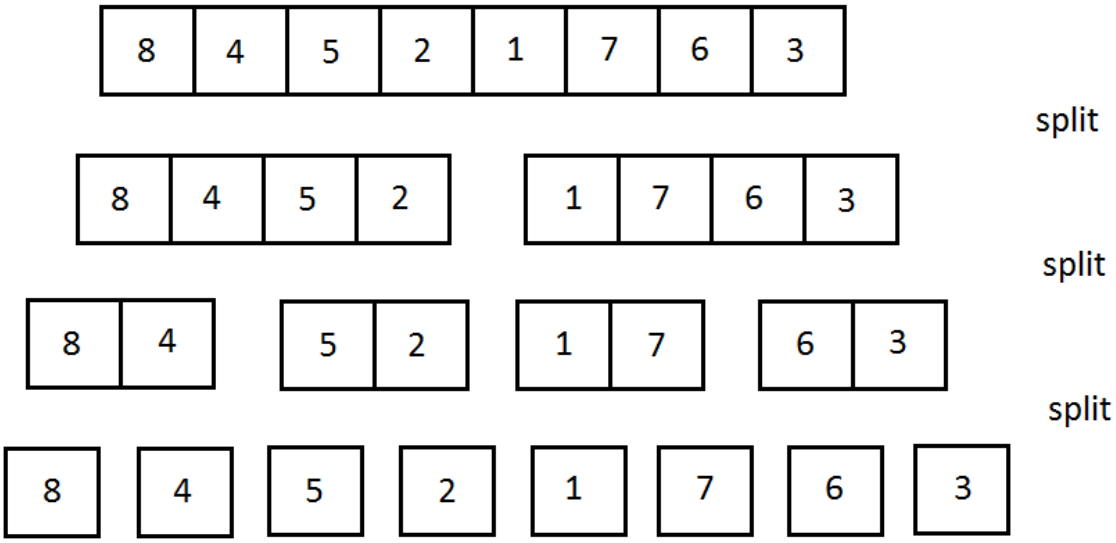
Sorting part II, mergesort and quicksort

Sorting: For mergesort, read chapter 2.3 and for quicksort, read chapter 7.1-7.2 in the text.

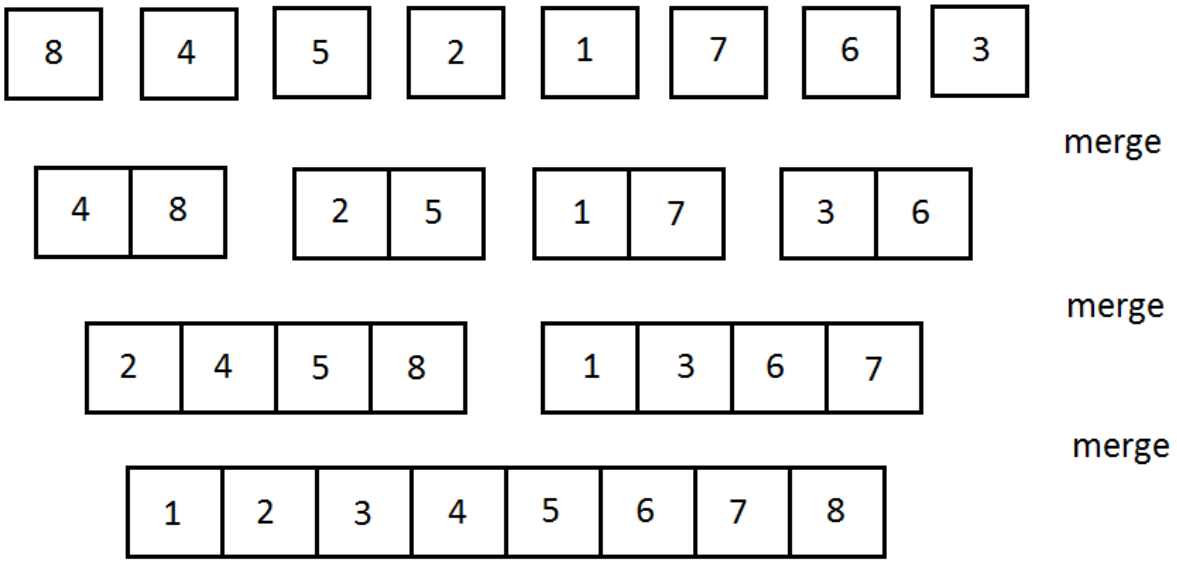
That worst case O(n^2) for sorting is too slow! Next, we'll discuss 3 algorithms that achieve O(n * log(n)) performance, which is much faster. These algorithms are heapsort, quicksort, and mergesort. We'll do mergesort and quicksort today. These manage to break the O(n^2) barrier in different ways.

1. Mergesort.

Mergesort takes a very simple approach to sorting, in which it keeps splitting its array in half until the array has only one item. A single-item array is, by definition, sorted; nothing is out of order there. This step is pretty easy to program.



When all the arrays are of size one, we begin merging them into size-2 arrays that are sorted. This is the part of the program that takes a little more work to write.

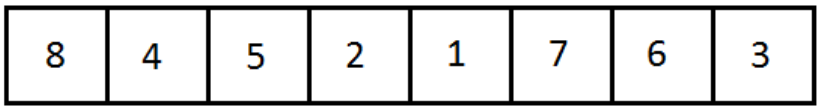


Mergesort splits a size-n array in half log2(n) times. It then merges each array; merging 2 arrays of size n/2 to get one array of size n takes n comparisons. Just as there were log2(n) splits, there are log2(n) merges. The total time for mergesort is n log2(n). This is true in the best case and the worst case (unlike insertion sort or bubble sort).

The downside of mergesort is that each of the little merges requires us to make an extra array to hold the merged data. We copy this array back at each step to the main array and write the merges into the second array. The overhead of this extra array is pretty expensive, for large n.

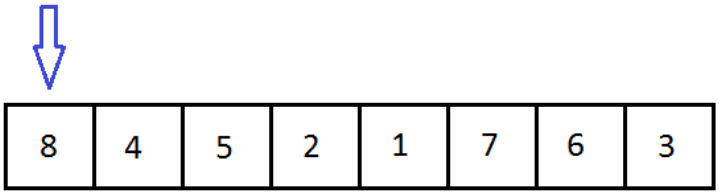
2. Quicksort

Quicksort takes a different approach to sort the array in place. Its first job is to identify a pivot element in the array. We'd like this element to be in the middle range of the array, so we often check the first element, the middle element, and the last element and take their median (middle) value. For us, that's 3. We'll remember that pivot value.

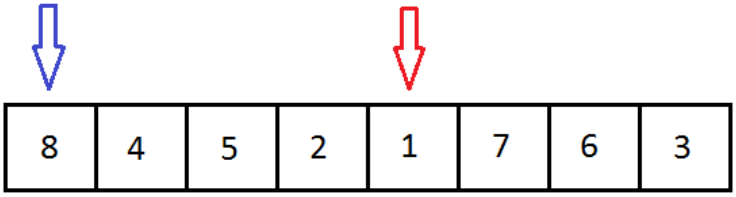


The main work of quicksort is a process called partitioning, which puts the pivot in the correct place and puts all the items <= pivot on the left hand side of the pivot, and all the items > pivot on the right hand side.

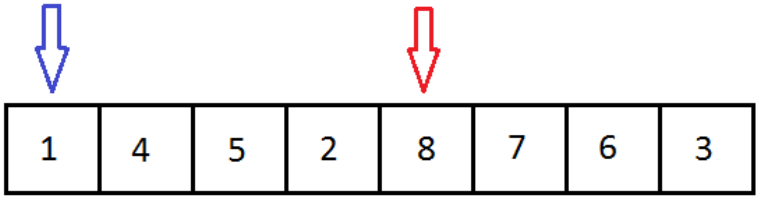
First in partition, we run a loop that starts at the front of the array and continues forwards as long as the items it finds are less than or equal to the pivot element. When we find an item that's bigger than the pivot, we will stop at that index. Here, that's the first item (the 8).



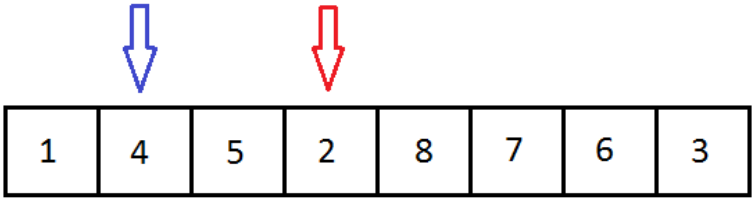
We also run a loop from the end of the array backwards that continues as long as the items it finds are greater than the pivot element. This loop stops at the 1, which is less than the pivot.



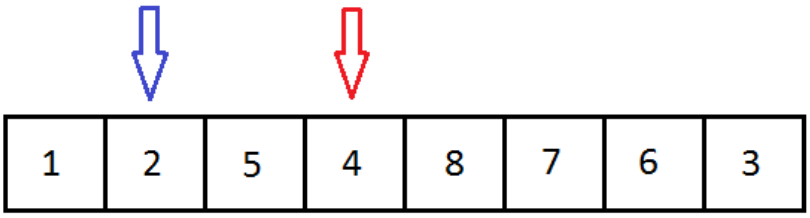
Now, we swap the 8 and the 1:



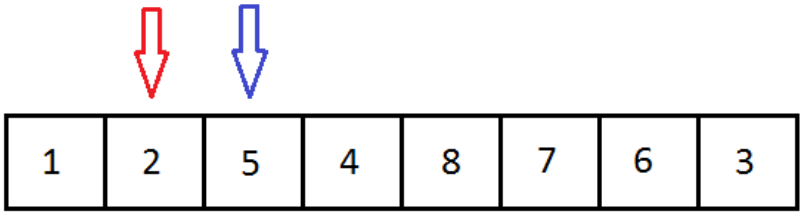
This leaves us free to continue the loops. The forward loop will now stop at the 4, and the backward loop will now stop at the 2:



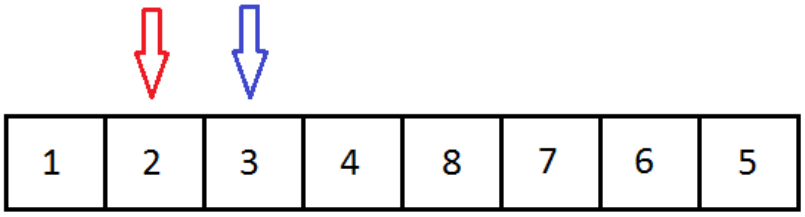
Then we swap these items:



Now, if we loop forward again until we find a too-big item, and backward again until we find a too-small item, our indexes cross:



This crossing tells us that we are nearly done with one partition. All that remains is to swap the pivot element, the 3, with the last item we found that was too big for the left hand side (the 5):



With this done, notice that the pivot element (the 3) is in place, and everything to the left of the pivot is less than or equal to the pivot, and everything to the right of the pivot is greater than the pivot.

What comes next? Recursively sort the subarrays [1 2] and [4 8 7 6 5] by partitioning each of these subarrays around a new pivot element. When all of these little arrays are sorted, the whole array is sorted as well.

```
quicksort(0, 1);
quicksort(3, 7);
```

One problem with the algorithm shown above is that it does not do well with duplicate pivots. (There were no duplicates in my example, which obscured this.) To handle duplicate pivots in reasonable time, you would modify the logic above to take items equal to the pivot into account in a partition.

```
Let pivot be the first item in the array a, a[0].
Set a counter lt = 0 and set a counter gt = size - 1;
Scan i from left to right using a for loop.
```

If (a[i] < pivot), exchange a[lt] with a[i], and increment both lt and i
Else if (a[i] > pivot), exchange a[gt] with a[i]; decrement gt
Else if (a[i] == pivot), increment i

Quicksort is usually O(n log n), but it can do worse. It depends on the pivot choice. What would partition have done if we'd chosen the 8 as the pivot? Or the 1? We make this choice carefully because if we're unlucky in every pivot selection, the algorithm degenerates to O(n^2) again.

Often, when your quicksort's sub-arrays become small, you sort them with something quadratic and easy, like insertion sort. We can get away with this as long as we only do it on arrays with size less than or equal to some constant.

We'll finish up the n log n recursive sorts next, with heapsort (chapter 6.1-6.5).