Quicksort

MergeSort, while it is clearly superior to the $O(n^2)$ sorting algorithms, suffers from two flaws

- the need to pass an additional temp array to the sorting algorithm. This is a pain at anytime, and could be a major problem if the array you are sorting is so big that you have trouble allocating another array of the same size.
- all of the copying back and forth between the data array and the temp array.

Quicksort is another divide-and-conquer algorithm that makes up for these deficiencies. You must be careful implementing QuickSort; small changes in the code can lead to incorrect sorts. When implemented correctly it is one of the best sorting algorithms known.

Quicksort was invented by the British computer scientist Tony Hoare in 1960. His original implementation was in Fortran (which didn't permit recursion) and the code was so complex most people couldn't follow it. Once recursion became available (in ALGOL 60) Hoare's recursive version was so simple most people didn't take it seriously.

Quicksort was a featured part of two very important standard systems: it was a C-language library function in the original Unix implementation, and it was the sorting algorithm used in the original Java implementation.

The idea behind QuickSort is simple. To sort the elements of array A between indices *first* and *last*:

a) Choose one of the elements of the array to be the "pivot" value. The algorithm works correctly regardless of which element you choose.
b) Partition (rearrange) the array so that elements that are less than the pivot come before it, and the elements that are greater than or equal to the pivot come after it. The pivot must then be in its final location. Suppose that is at index i.
c) Recursively sort the elements between *first* and *i*-1, and the elements between *i*+1 and *last*.

If there are n elements in the list the rearranging can be done in one pass with at most n steps.

Note that while the algorithm will correctly sort regardless of which pivot value is chosen, the choice of pivot does make a difference.

Suppose the pivot value we choose happens to be the largest element. Then one of the recursive calls gets an array with no elements and the other gets all but one of the elements. If the array happens to be already sorted and we choose the leftmost entry to be the pivot, this will happen at each step and we end up doing n recursive calls, sorting n elements, (n-1) elements, (n-2) elements, and so forth element. The partitioning process takes n + (n-1) + (n-2) + ... steps, which altogether gets us $O(n^2)$ steps.

So in the worst case QuickSort is $O(n^2)$. If that was the end of the story we wouldn't be interested in it.

Think about the more typical case -- unless we are really unlucky, most of the time the pivot value should be around the middle of the data. This means that the recursive calls will each be to about half of the data. Just as with MergeSort, we would expect QuickSort to run in time O(n*log(n)) in the typical case. Even better than MergeSort, QuickSort doesn't have to copy all of the data on each step.

There are some things we can do to guard against especially unfavorable cases. Sorted, reverse-sorted, and nearly-sorted data all occur more often than you might think; using the first or last element of the list as the pivot doesn't do well in these cases. An easy remedy for this is to use the middle of the array as the pivot value. Some authors recommend taking the median of the left end, the right end, and the middle value. Steps like this don't avoid the worst-case $O(n^2)$ behavior, but they insure that the algorithm reaches its worst case only in very specific and unlikely situations.

The partition step of the algorithm needs to be coded carefully but the idea behind it is easy.

Suppose we start with the following data and choose as the pivot the middle value:

[23 3 9 7 14 8 7 12 6]

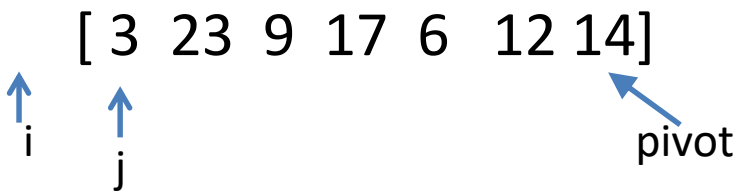
pivot

Start by interchanging the pivot with the last element, so it is out of the way. Let *first* and *last* be the first and last indices in the portion of the array we are partitioning.

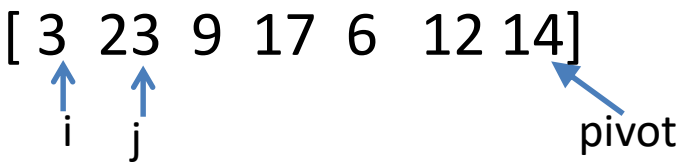
Now we have a loop that keeps track of variables *i* and *j*. Variable *j* just marches up from *first* to *last* in a for-loop. All of the data from *first* up to and including index *i* is less than the pivot; everything to the right of index *i* up to *j* is greater than or equal to the pivot.

Initially we haven't found any data less than the pivot, so we set *i* to *first*-1.

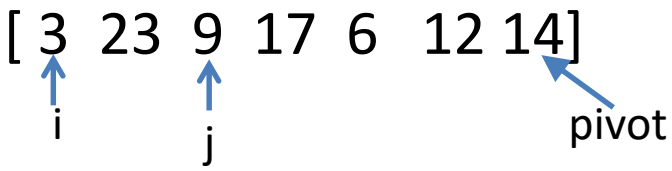
Each time we find a value less than the pivot we increment *i* (into the values greater than the pivot) and swap the large value at index *i* with the small value we have just discovered at index *j*.



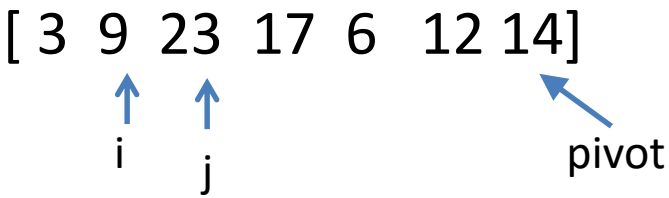
The value at index j is less than the pivot, so we increment i, swap the values at indices *i* and *j* (which does nothing because i and j are the same index) and move *j* to the next element



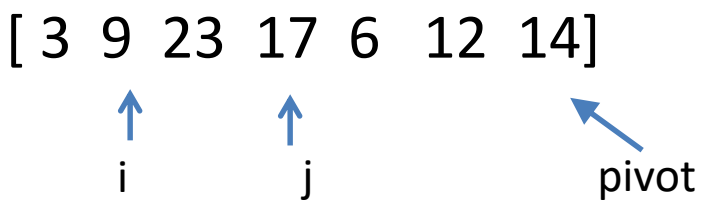
The value at *j* is greater than the pivot so we do nothing; j moves to the next element



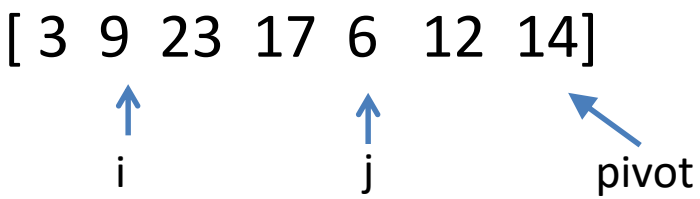
This time the value at *j* is less than the pivot so we increment i to make room, swap the values at *i* and *j*:



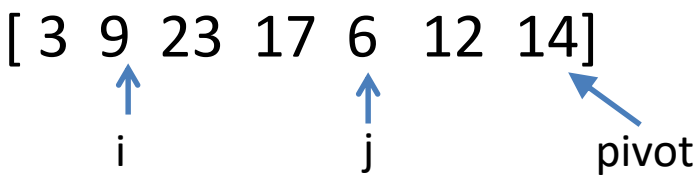
Notice that everything at *i* and to its left is less than the pivot, everything to the right of *i* up to *j* is greater or equal to the pivot. We increment *j*.



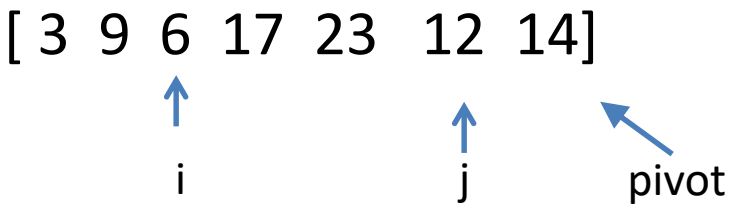
The value at *j* is greater than the pivot, so we just increment *j*.

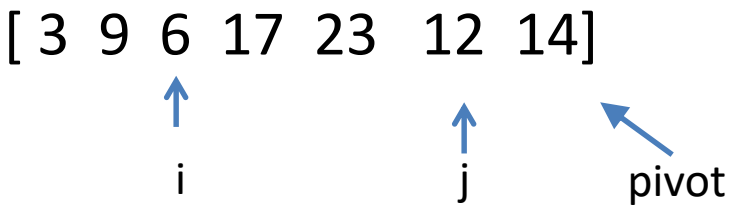


Still everything at i and to its left is less than the pivot; everything past *i* and before *j* is greater.

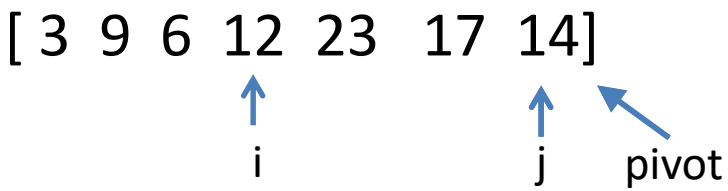


The value at *j* is less than the pivot so we increment *i* to make room, swap the values at *i* and *j*, and then increment *j*:

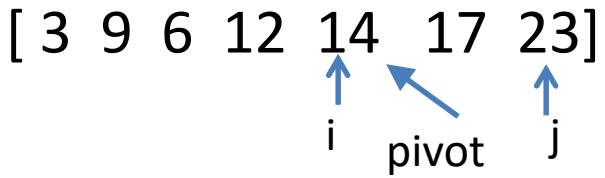




Once more the value at index *j* is less than the pivot so we increment *i* to make room, swap the values at *i* and *j*, and increment *j*.



We have arrived at the end of the for-loop. The only step that remains is to put the pivot in its final location. Everything at *i* and to its left is less than the pivot, so we swap the pivot with the value at index i+1:



You can see that the pivot is in the right location and it separates the array values into those less than the pivot and those greater or equal to the pivot.

The next two slides have the code for Quicksort.

```
private static <E extends Comparable<? super E>> void
        QuickSort(E[] A, int first, int last) {
                if (first < last) {
                        int mid = (first+last)/2;
                        swap(A, mid, last);
                        int pivotIndex = partition(A, first, last);
                        QuickSort(A, first, pivotIndex-1);
                        QuickSort(A, pivotIndex+1, last);
                }
        }
```

```
private static <E extends Comparable<? super E>>
int partition(E [] A, int first, int last) {
    E pivot = A[last];
    int i = first-1;

    for( int j=first; j < last; j++) {
        if (A[j].compareTo(pivot)  < 0){
            i += 1;
            swap(A, i, j);
        }
    }
    i += 1;
    swap(A, i, last);
    return i;
}
```