Sorting

Carol Zander

$O(n^2)$ algorithms include bubble, insertion, and selection. More efficient algorithms are typically recursive.

Quick sort (Hoare sort)

QuickSort (HoareSort in honor of Tony Hoare, its creator in 1960) is the **most commonly used sorting algorithm for large arrays** because it is (usually, meaning on average) both **time efficient (like you will see with merge sort)** and **space efficient (unlike merge sort).** It is a clever algorithm, uses a **divide-and-conquer approach**. In this sort, choose one particular element that is called the "pivot." The array is then divided into all elements that are less than the pivot (placed at the array beginning) and all elements that are greater than (or equal to) the pivot (placed at the array end). The pivot is placed in its sorted position. The algorithm is then recursively called on the smaller array segments. There are variations of this sort, mainly how the pivot is chosen and whether the pivot has to be an element in the array, but the algorithm is essentially the same.

```
//---------------------------------------------------------------------------
// median - find median of low, center, high values so that a[low] ends up
// as the value < median; a[high] is the value > median; a[center] is the
// median. Hide the median, the pivot, at the position just before the array
// end (which is larger than the pivot and in an acceptable partitioning place)

int median(int a[], int low, int high) {
   int center = (low+high)/2;
   if(a[low] > a[center])  swap(a[low], a[center]);
   if(a[low] > a[high])    swap(a[low], a[high]);
   if(a[center] > a[high]) swap(a[center], a[high]);
   swap(a[center], a[high-1]);                      // hide the pivot
   return a[high-1];                                // return pivot
}

//---------------------------------------------------------------------------
// partition - separate array into 2 partitions: numbers < pivot, and > pivot
void partition(int a[], int pivot, int& up, int& down) {
   for (;;) {
      while (a[++up] < pivot);            // find value > pivot
      while (a[--down] > pivot);          // find value < pivot
      if (up < down)
         swap(a[up], a[down]);
      else
         break;
   }
}

//---------------------------------------------------------------------------
// quicksort - sort an array.  continually partition into sets of
// numbers < pivot and numbers > pivot. The pivot is put into its rightful
// sorted position. Recursively continue.

void quickSort(int a[], int low, int high) {
   int up, down, pivot;

   // use quicksort unless the size is <= CUTOFF, then use insertion sort
   if (low+CUTOFF <= high) {
      pivot = median(a, low, high);
      up = low;
      down = high-1;

      // separate array into 2 partitions: numbers < pivot, numbers > pivot
      partition(a, pivot, up, down);
      swap(a[up], a[high-1]);              // put pivot in rightful position
      quickSort(a, low, up-1);
      quickSort(a, up+1, high);
   }
   else
      insertionSort(a, low, high-low+1);
}
```

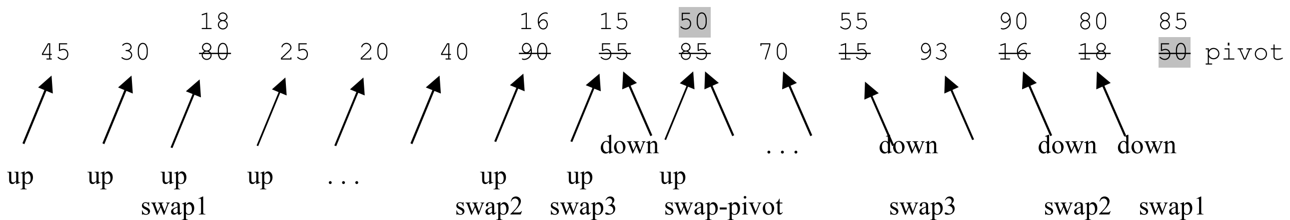
You want to choose a pivot that is as close to the middle of the range of numbers as possible, but you don't want to do too much work. The scheme above finds the median of the first, middle, and last elements. Your text suggests taking the first element. This is risky because if the array is sorted, you end up with a running time as bad as the $O(n^2)$ algorithms. The goal is to find a pivot roughly in the middle of the range of numbers.

To partition the array, one counter walks up the array, another walks down the array. When walking up, you look for an item larger than the pivot, when walking down, you look for an item smaller than the pivot. Swap them and continue the walking up and down and swapping until the up and down counters cross.

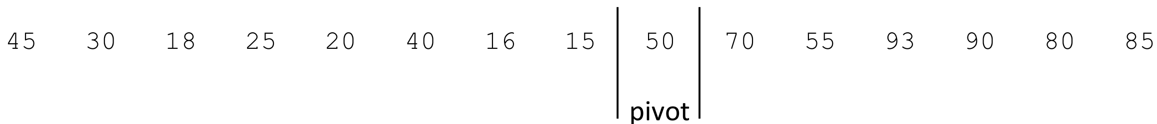
This is not 100% what the code does because when finding the median, the values are not always swapped in my example. If that is done, the numbers get sorted too quickly and you don't get a feel for the algorithm. We would need a much larger number of numbers.

Find the pivot and get it out of the way by putting it at the end.

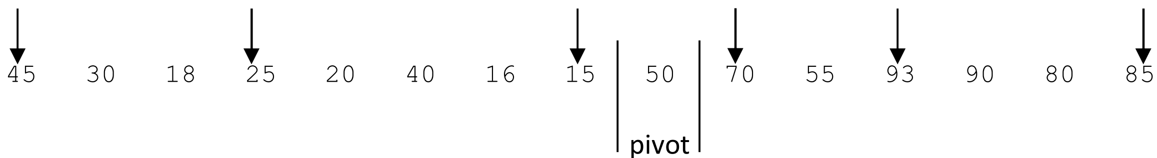
Then walk up the array, looking for a number larger than the pivot, walk down, looking for a number smaller than the pivot and swap them. This is one pass of the algorithm.



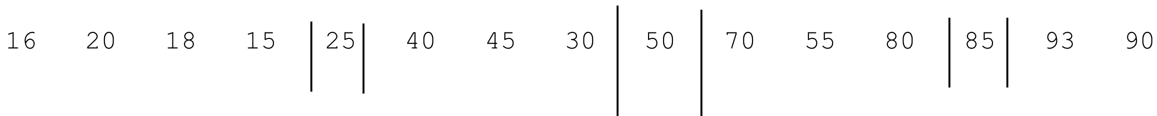
When the up and down cross, swap the pivot with the element at subscript *up*.



Do the same thing on each "half". For the bottom half, the pivot is 25. The upper half pivot is 85. Again, when finding the median, the numbers are not always swapped (although the pivot is hidden at the end).



After the up and down swapping, the array is shown.



Merge sort

Merge sort is another divide-and-conquer algorithm. In general, merge sort works in a manner similar to hoare sort -- splits the data into two subsets, solves the problem on the smaller sets, and then combines the sets again. Merge sort is more straightforward -- breaks the array into two subsets by cutting it in the middle, sorts both subsets (recursively), and then combines them again.

```
//-------------------------------------------------------------------------
// merge
// merge two sorted arrays into one long sorted array

void merge(int a[], int low, int mid, int high) {
   int temp[high - low + 1];

   int low1 = low;
   int high1 = mid;
   int low2 = mid + 1;
   int high2 = high;

   // As long as both lists still have elements, add the next.
   int index;
   for (index = 0; (low1 <= high1) && (low2 <= high2); index++) {
      if (a[low1] < a[low2]) {
         temp[index] = a[low1];
         low1++;
      }
      else {
         temp[index] = a[low2];
         low2++;
      }
   }

   // One of the lists still has elements, so add them now.
   for (; low1 <= high1; index++, low1++)
      temp[index] = a[low1];

   for (; low2 <= high2; index++, low2++)
      temp[index] = a[low2];

   // Copy back into the array.
   for (index = 0; index < high - low + 1; index++)
      a[index + low] = temp[index];
}

//-------------------------------------------------------------------------
// mergeSort
// Break the array into two subsets by cutting it in the middle,
// sorts both subsets and combine

void mergeSort(int a[], int low, int high) {
   if (low < high) {
      int mid = (low + high) / 2;
      mergeSort(a, low, mid);
      mergeSort(a, mid + 1, high);

      merge(a, low, mid, high);
   }
}
```

When executing the merge sort, on the way down of the recursion, the items are split into parts, but the actual work of merging the smaller sets into sorted bigger sets is done on the way back up from the recursion. These execution trees show calling the recursive mergeSort, the "down", and the returning, the "up", separately.