That's the Way It Goes

How to sort: the good, the bad, the ugly

Algorithms Alfresco

by Julian Bucknall

I was sitting back in the audience seats at a rehearsal for *Alice in Wonderland* recently, feeling kind of bored. My character, Lewis Carroll, didn't have to come on for a while, and I was watching a dance routine being rehearsed with the cards: the three who are busy painting the white rose bushes red (having planted the wrong sort) before the Queen of Hearts comes by. Part of my mind was trying to think of a topic for this month's column. Suddenly I realised that the cards (actually three actors with six foot tall thin foam mattresses) were shuffling themselves, under the supervision of the choreographer, trying to work out a set of moves to get them in order again. *Ping!* The topic came into my mind: sorting algorithms, and what better way to illustrate them than with cards.

So arm yourself with a pack of cards and, in honour of *Alice*, deal yourself all of the Hearts. When I describe an algorithm, and we'll be visiting a few, try it out with your hand of cards. This way you'll be able to more easily visualise what's going on.

Love Child

The first sorting routine people learn about, probably at their grandmother's knee, is the bubble sort. It's quite easy to implement, but unfortunately, it's also the worst in terms of efficiency. We'll look at it briefly just so that you can steer clear of it in the future.

Take your hand of Hearts, and look at the right two cards, the 12th and 13th. Swap them over, if needed, to put them in ascending order. Look at the 11th and 12th. Swap them over, if required, to put them in ascending order. Continue with the 10th and 11th cards, the 9th and 10th cards and so on. Once you've got to the first and second cards, and swapped them to get them in order if required, you'll have the Ace of Hearts in the last, leftmost position (we're assuming the pip value of the Ace is one). If you observe what's happening as you do this, you'll see that once you've 'snagged' the Ace, it 'bubbles' along until it reaches the left hand side.

Now start over from the right hand side, and do the same thing until you get to the second card and you'll have the deuce (the two) in position. Start over yet again until you reach the third card and the three of Hearts will be in position. Continue, 9 more times, each time reducing the number of cards you check. That's bubble sort. It's pretty dreadful, as you can see. Cards are continually being compared and swapped with each other, the lowest ones being caught up and slowly moving along to their correct position. For a small number of cards, it's not so bad, but just imagine trying to sort 52 cards by this means.

Listing 1 (if you really want to look at it, that is) is the bubble sort. At this point a quick note is necessary on the source for this article. All the routines assume some arbitrary type called `TSortElement` for the elements. This can be an `integer`, `double`, or something like a record. All the sorts take an array of these variables, a left and a right index to sort between (inclusively), and a function that takes two elements as parameters and returns whether the first is less than the second. The sort routines all order the elements in the array *in situ*.

➤ Listing 1: The Bubble sort

```
procedure BubbleSort(var aItemArray : array of TSortElement;
  aLeft, aRight : integer; aLessThan : TLessFunction);
var
  i, j : integer;
  Temp : TSortElement;
begin
  for i := aLeft to pred(aRight) do
    for j := aRight downto succ(i) do
      if aLessThan(aItemArray[j], aItemArray[j-1]) then begin
        Temp := aItemArray[j];
        aItemArray[j] := aItemArray[j-1];
        aItemArray[j-1] := Temp;
      end;
end;
```

There is a little known variant of bubble sort that works better: the shaker sort. Start off as with the bubble sort. Once the Ace is in position, instead of starting over from the right hand side as we did with the bubble sort, start over from the left. Compare the second and third cards, swap them if needed. Do the same with the 3rd and 4th, the 4th and 5th, and so on until you reach the end. You'll have snagged the King *en route* and it will be in the 13th position. Now, go back from the right, comparing and swapping until you get the deuce in position. Continue going back and forth until you find the seven is in position.

As you may gather from the description, coding a shaker sort is a little more involved. Listing 2 has the details. Like the bubble sort I won't go through the details: there are better sorts to use, so the shaker sort is somewhat of academic interest.

Breakout

The next sort we'll consider is the selection sort. This is a fairly good sort in certain circumstances and is much easier to understand. Starting at the right, scan the cards until you find the Ace. Swap it with the first card. Scan the cards from the right again, until you find the deuce. Swap it with the second card. Scan from the right again until you find the three. Swap it with the third card. Continue the same process with the four, five, six, etc. You'll find that you have to loop 12 times through the hand to get all the cards in their proper positions (the 13th loop is not required, since obviously the King will be in the correct place already).

If you think about it a little, this is almost like the bubble sort except we drastically reduce the number of swaps we do. We identify the smallest and *then* we do the swap. With bubble sort we're swapping as we're scanning.

Obviously we took a short cut in our example with selection sort: in real life we don't know which is the smallest card in our hand. We have to scan though all the cards from the right up to the current end point to find the smallest. Apart from that, the Delphi routine follows the physical card sorting pretty well. See Listing 3 for the routine.

I mentioned that this sort was "good in certain circumstances". The thing to note about selection sort is that there are a maximum of *n*-1 swaps to make to get the *n* cards into their correct positions. If the elements that we're sorting are large (and hence swapping them is time consuming or, to put it another way, swapping two elements is very expensive time-wise compared with comparing them) this sort is in fact the best we can do. The time to perform a selection sort is otherwise dominated by the comparisons we have to do.

```
procedure ShakerSort(var aItemArray : array of TSortElement; aLeft, aRight :
  integer; aLessThan : TLessFunction);
var
  i : integer;
  Temp : TSortElement;
begin
  while (aLeft < aRight) do begin
    for i := aRight downto succ(aLeft) do
      if aLessThan(aItemArray[i], aItemArray[i-1]) then begin
        Temp := aItemArray[i];
        aItemArray[i] := aItemArray[i-1];
        aItemArray[i-1] := Temp;
      end;
    inc(aLeft);
    for i := succ(aLeft) to aRight do
      if aLessThan(aItemArray[i], aItemArray[i-1]) then begin
        Temp := aItemArray[i];
        aItemArray[i] := aItemArray[i-1];
        aItemArray[i-1] := Temp;
      end;
    dec(aRight);
  end;
end;
```

➤ *Listing 2: The Shaker sort*

```
procedure SelectionSort(var aItemArray : array of TSortElement; aLeft, aRight :
  integer; aLessThan : TLessFunction);
var
  i, j : integer;
  IndexOfMin : integer;
  Temp : TSortElement;
begin
  for i := aLeft to pred(aRight) do begin
    IndexOfMin := i;
    for j := succ(i) to aRight do
      if aLessThan(aItemArray[j], aItemArray[IndexOfMin]) then
        IndexOfMin := j;
    Temp := aItemArray[i];
    aItemArray[i] := aItemArray[IndexOfMin];
    aItemArray[IndexOfMin] := Temp;
  end;
end;
```

➤ *Listing 3: The Selection sort*

```
procedure UsualInsertionSort(var aItemArray : array of TSortElement;
  aLeft, aRight : integer; aLessThan : TLessFunction);
var
  i, j : integer;
  Temp : TSortElement;
begin
  for i := succ(aLeft) to aRight do begin
    Temp := aItemArray[i];
    j := i;
    while (j > aLeft) and aLessThan(Temp, aItemArray[j-1]) do begin
      aItemArray[j] := aItemArray[j-1];
      dec(j);
    end;
    aItemArray[j] := Temp;
  end;
end;
```

➤ *Listing 4: The Insertion sort*

Better Make It Better

The next sort on our agenda is the insertion sort. If anyone has played whist or bridge, they certainly will be familiar with this one. We start from the left this time. Compare the first two cards. Put them in order. Look at the third card. Insert it into the right place among the first two (of course, it may already be in the correct position). Look at the fourth card and insert it into the correct position in the three cards that are already sorted. Continue with the 5th, 6th, etc, cards. You'll notice that the cards on the left of the card being considered are always sorted.

Listing 4 shows my first implementation of this sort. We shall put aside our cards for a moment, because we can fiddle with the implementation of this algorithm to make it a little more efficient.

If you look at the code you'll see that the inner loop, where we are trying to find a place to insert our current item, stops with one of two conditions. The first condition is that we manage to find an element that is less than the one we are trying to insert. (At that point we know where to insert our element: just after the smaller element we found.) The second condition is that we've run out of elements: the item we're trying to insert is *smaller* than all the currently sorted elements. We need to insert our item before *all* the other sorted elements. But note that this second condition is tested with every pass through the loop, but it's *only* used when we've managed to scan through all the elements and we need to stop falling off the end. Traditionally, the way to get rid of this extra check per loop is to have a *sentinel* element on the end, one that is less than all the other elements. I've always found this to be a cop-out in real programming life. Firstly, sometimes you don't know what the smallest value might be (quick, what is the smallest number that can be represented by a variable of type `Double`?) and secondly, sorting routines are generally written to accept an array and a number of elements and you have to sort the array *in situ*. Where are you going to put this extra element? You'd have to allocate an array that is one larger in size than the array you're passed, set the first element to this smallest value, copy over all the other elements from the array you're passed, sort them, and then copy all the elements back. Brrrr, thanks, but no thanks.

An alternative is to have a presort pass where you find the smallest element in the set you're given, put that into the first position (essentially, the first pass is with a selection sort), and then perform the insertion sort without worrying about falling off the end. Listing 5 shows this version of insertion sort.

Before moving on, we should discuss the performance characteristics of these sorts. We briefly mentioned a performance characteristic of the selection sort (it's great if element moves take a long time), but let's be a little more precise. The time to execute a sort of the type discussed so far is proportional either to the number of element comparisons, to the number of element moves or exchanges, or both. All these sorts are quadratic-time algorithms, both in the worst case (the elements are reverse-sorted to begin with) and in the average case. In other words, if it takes *x* milliseconds to sort 100 elements, it'll take 100*x* milliseconds to sort 10 times as many elements, or 1000 of them. As you can see the time taken to do these sorts increases by leaps and bounds the more elements there are to sort. Although sorting a small number of elements would be virtually instantaneous, with a larger number it would give you time to go get a cup of coffee.

Having said that, what happens if one of these sorts operates on a set of elements that's 'almost' sorted? Consider how the sorts would work on a completely sorted file. Insertion sort is probably the most efficient: it makes one pass through the set and determines that all elements are in their correct place. Bubble sort and shaker sort and selection are all about as bad as each other: they make a complete pass through the set and for each element they make a comparison pass through the remaining elements. If we modify bubble and shaker sort to exit once a pass occurs without any swapping of elements, they too can 'sort' an already sorted set in one pass. Selection sort has no simple way to find out if the file is sorted. On an almost sorted set of elements, research has shown that a modified bubble or shaker sort or an insertion sort is almost linear in time. The best that can be said for selection sort is the conclusion we came to before: if the time to move an element is large compared to the time to compare two elements, then selection sort is almost linear (for N elements there will be N-1 exchanges at most and that time will swamp the $N^2/2$ comparisons at worst.)

I would like to reinforce the fact that, if all the elements are either in the correct position or at most a few elements away from their correct position, insertion sort is the king as it runs in linear time. We'll be using this fact in a moment.

With these four sorts out of the way, we've completed looking at the so-called elementary sorts. The next set of sorts (shellsort, merge sort and heap sort) are more advanced. I won't deal with merge sort or heap sort in this article: merge sort because it deserves an article to itself and heap sort because it relies on a data structure called a *priority queue* that we'll also deal with at another time. Shellsort is fair game though and doing it with cards will take some doing.

Fooled By A Smile

Shellsort is named after its inventor, Donald L Shell, and at first I found it a little bizarre; most probably because I was trying to see the 'shells' (my first exposure to it didn't explain the name). Look back at the standard insertion sort (rather than our optimised one). Imagine that the smallest element is found at the end of the unsorted set. How many exchanges are required to get this element to its

➤ *Listing 5: The better Insertion sort*

```
procedure InsertionSort(var aItemArray : array of TSortElement; aLeft, aRight :
  integer; aLessThan : TLessFunction);
var
  i, j : integer;
  IndexOfMin : integer;
  Temp : TSortElement;
begin
  {find the smallest element and put it in the first position}
  IndexOfMin := aLeft;
  for i := succ(aLeft) to aRight do
    if aLessThan(aItemArray[i], aItemArray[IndexOfMin]) then
      IndexOfMin := i;
  if (aLeft <> IndexOfMin) then begin
    Temp := aItemArray[aLeft];
    aItemArray[aLeft] := aItemArray[IndexOfMin];
    aItemArray[IndexOfMin] := Temp;
  end;
  {now sort via insertion method}
  for i := aLeft+2 to aRight do begin
    Temp := aItemArray[i];
    j := i;
    while aLessThan(Temp, aItemArray[j-1]) do begin
      aItemArray[j] := aItemArray[j-1];
      dec(j);
    end;
    aItemArray[j] := Temp;
  end;
end;
```

rightful position? Well, it'll happen right at the end of the sort, and it'll be swapped over with every single element as the algorithm drags it all the way to the start of the set. Shellsort is a variant of insertion sort that attempts to move well-out-of-place elements to their rightful positions by jumping over intermediary elements.

Back to the cards. Doing this in your hands is a little difficult, so shuffle the Hearts and deal them out in one row. Push up the first card and every fourth card from that point (ie, you'll push up the 1st, 5th, 9th and 13th cards). Insertion sort these four cards. Push them back down again. Now push up every 4th card starting with the second card (ie, the 2nd, 6th and the 10th cards). Insertion sort these three. Continue this process until you reach the 5th card, at which point you've completed this first stage. The cards are now said to be in *4*-sorted order. Whichever card you select, the subset generated by counting 4 cards forwards and backwards is sorted. Note the set as a whole is still not sorted, but every card has been visited at least once. No matter how hard you shuffled, cards are now in the vicinity of where they should be, large jumps have been made in exchanges. Now we perform a standard insertion sort through the set of cards. That in essence is the Shellsort.

To be more rigorous, the Shellsort works by insertion sorting subsets of the main set. Each subset is obtained by taking every *h*th element starting at any position in the original set. There will be *h* subsets to independently sort. Once the set of elements is *h*-sorted, we reduce *h* to some new value and then *h*-sort the set with this new value. We continue until we reach the value 1 for *h* and we perform a standard insertion sort (which could be called *1*-sorting the set). The essence of Shellsort is that *h*-sorting with large values of *h* causes elements to jump into the vicinity of their correct places quickly. As we reduce *h*, we find elements migrate quickly to their correct positions, and the final

```
procedure ShellSort(var aItemArray : array of TSortElement; aLeft, aRight :
  integer; aLessThan : TLessFunction);
var
  i, j : integer;
  h    : integer;
  Temp : TSortElement;
begin
  {firstly calculate the first h value we shall use: it'll be about
   one ninth of the number of the elements}
  h := 1;
  while (h <= (aRight - aLeft) div 9) do
    h := (h * 3) + 1;
  {start a loop that'll decrement h by one third each time through}
  while (h > 0) do begin
    {now insertion sort each h-subfile}
    for i := (aLeft + h) to aRight do begin
      Temp := aItemArray[i];
      j := i;
      while (j >= (aLeft+h)) and aLessThan(Temp, aItemArray[j-h]) do begin
        aItemArray[j] := aItemArray[j-h];
        dec(j, h);
      end;
      aItemArray[j] := Temp;
    end;
    {decrease h by a third}
    h := h div 3;
  end;
end;
```

➤ *Listing 6: The Shell sort*

1-sort cleans everything up without elements moving much at all.

I'm sure you'll agree this is a bit too much hand-waving. What values of *h* do we use? Well, this is difficult to say. Shell, in his original paper, suggested the sequence 1, 2, 4, 8, 16, 32, etc (in reverse, obviously) but this suffers badly in the worst case because odd numbered elements are never compared to even-numbered elements until the final pass. If you managed to get the smallest half of the elements in the odd positions and the largest half in the even positions, you'd still have a lot of movement to do in that final pass.

The mathematics to prove good characteristics for a given sequence is extremely difficult for such a simple algorithm, and indeed there may be better sequences for *h* than have been devised so far. Unfortunately no one has yet managed to analyze Shellsort, thus comparing Shellsort with other sorts mathematically is difficult. We are forced to rely on statistical methods to analyze different sequences and against other sorts.

D E Knuth proposed the sequence 1, 4, 13, 40, 121, etc (each value is one more than three times the previous). This has good performance characteristics for moderately large sets as well as being simple to calculate. We use Knuth's sequence in the code in Listing 6. Another sequence that is better for large N is 1, 8, 23, 77, etc (ie, $4^{i+1}+3\cdot 2^{i}+1$ for i>=0) but it's a little more complex to calculate.

Compared with the sorts we've seen so far, Shellsort is relatively stable in terms of best and worst execution profiles. It is remarkably difficult (mainly because Shellsort has not been analysed fully) to devise a set that causes Shellsort with a given sequence to revert to quadratic time. For this reason, and for the fact that it's easy to code (especially with Knuth's sequence), Shellsort makes a good first choice for sorting moderately large to large sets. We'll be visiting quicksort in a moment, which is especially faster with large sets, but it is harder to get quicksort to behave with certain ordered sets, and getting a correct version of quicksort requires considerably more code, and more complex code at that.

Surrender

And so we reach the well-known quicksort. We'll be spending some time on this sort variant so go get a quick cuppa first and get your cards.

Shuffle and deal the Hearts into a line. Choose the end card, the thirteenth. What we will be doing is to *partition* the cards so that all cards less than the thirteenth appear to its left and all greater cards are to its right. We then perform the same process on each of these two sets of cards, the lesser set and the greater set, and so on, so forth. Right, here goes. Start from the left, and find the first card that is greater than or equal to our card. Starting from the right (ie, the twelfth card), find the first card that is less than or equal to our card. These two cards are out of place, so we swap them over. Continue from where we left off on the left and find the next card that is greater or equal to than our card. Do the same from the right finding the next lesser or equal to and swap over. Eventually we'll find that our scanning from the left and right cross over. Swap over the card at which we crossed (it'll be larger than our card) with our card, the thirteenth. We now have all cards less than the one we're interested in on its left and all those greater than ours on its right. Push up this card (just to note where we are). Take the left hand set, and do the same thing. Continue in this way until the set of cards we're trying to partition has no cards in it, or just one (which, by definition almost, is bound to be sorted). Now consider the next unsorted set on the right. And perform the same process.

To help you along, here's an example with cards I dealt recently (I've marked the Ace as 1, the Jack as J, Queen as Q, and King as K). I dealt the following:

```
J 1 5 2 Q K 9 10 7 4 8 3 6
```

For the start, we'll choose the 6 as our partition card. Scan from the left until we find a card greater than 6: the Jack. Scan from the right until we find a card less than the 6: the 3. Swap them over (I've marked the cards that moved in red)

```
3 1 5 2 Q K 9 10 7 4 8 J 6
```

Continue. From the left we reach the Queen, from the right we reach the 4. Swap them over:

```
3 1 5 2 4 K 9 10 7 Q 8 J 6
```

Continue. From the left we reach the King, from the right we reach the 4, *but we crossed over*. Swap the King and the 6, the card we selected in the first place. Notice that everything less than 6 is to its left and everything greater is to its right; hence the 6 must be in the correct place. We'll underline the 6 to indicate it's in the right position.

```
3 1 5 2 4 6 9 10 7 Q 8 J K
```

Take the left hand set and do the same. The 4 is the card *du jour*. Scan from the left, the first card greater than 4 is the 5; from the right, the first smaller is the 2. Swap them over.

```
3 1 2 5 4 6 9 10 7 Q 8 J K
```

Scan from the left for a greater card, we get the 5 again. From the right, we hit the 2, but we crossed over. So swap the 5 and the partition card, the 4. The 4 is now in its correct place and is underlined.

```
3 1 2 4 5 6 9 10 7 Q 8 J K
```

Take the new left hand set and do the same. The partition card is the 2. Scan from the left for the first greater (the 3) and from the right for the first lesser (the 1). Swap them over.

```
1 3 2 4 5 6 9 10 7 Q 8 J K
```

Scan from the left, we hit the 3 and scanning from the right we cross over. So swap the 3 with our partition card, the 2. It's now in the right place, so underline it.

```
1 2 3 4 5 6 9 10 7 Q 8 J K
```

Take the new left hand set: it has one card, therefore it must be in the right place. Underline it.

```
1 2 3 4 5 6 9 10 7 Q 8 J K
```

Take the next set from the left. Again it has one card (the 3), so is in the correct place. The next set along *also* has one card. At this point, our cards look like this:

```
1 2 3 4 5 6 9 10 7 Q 8 J K
```

Now apply this process to the remaining subset (the one on the right from the original partition). Here's the sequence of 'moves' (although note that a lot of times no swapping takes place):

```
1 2 3 4 5 6 9 10 7 8 Q J K
1 2 3 4 5 6 9 10 7 8 J Q K
1 2 3 4 5 6 7 10 9 8 J Q K
1 2 3 4 5 6 7 8 9 10 J Q K
1 2 3 4 5 6 7 8 9 10 J Q K
1 2 3 4 5 6 7 8 9 10 J Q K
1 2 3 4 5 6 7 8 9 10 J Q K
1 2 3 4 5 6 7 8 9 10 J Q K
```

Low Down Dirty Business

Quicksort, then, is an example of a divide and conquer technique, and is innately recursive. Also from watching me play cards, or from your own experiments, it's also relatively tedious once subsets get quite small.

The version of quicksort I demonstrated above is also *extremely* bad when the original set was sorted: the subsets always appear on the left, and reduce in size by one each time, the minimum allowed. We will be discussing

➤ *Listing 7: The quicksort recursive routine*

```
procedure UsualQuickSort(var aItemArray : array of TSortElement; aLeft, aRight :
  integer; aLessThan : TLessFunction);
  procedure QuickSortPrim(L, R : integer);
  var
    DividingItem : integer;
  begin
    {stop the recursion, if needed}
    if (R - L) <= 0 then
      Exit;
    {otherwise, partition about the final element in the set}
    DividingItem := Partition(L, R);
    {recursively quicksort the two subsets either side of the dividing
     element}
    QuicksortPrim(L, pred(DividingItem));
    QuicksortPrim(succ(DividingItem), R);
  end;
begin
  {start it all off}
  QuicksortPrim(aLeft, aRight);
end;
```

```
function Partition(L, R : integer): integer;
var
  i, j : integer;
  Last : TSortElement;
  Temp : TSortElement;
begin
  {set up the indexes}
  i := L;
  j := pred(R);
  {get the partition element}
  Last := aItemArray[R];
  {do forever (we'll break out of the loop when needed)}
  while true do begin
    {find the first element greater than or equal to the partition
     element from the left; note that our partition element will
     stop this loop}
    while aLessThan(aItemArray[i], Last) do inc(i);
    {find the first element less than the partition element from the
     right; check to break out of the loop if we hit the left
     element - we have no sentinel there}
    while aLessThan(Last, aItemArray[j]) do begin
      if (j = L) then Break;
      dec(j);
    end;
    {if we crossed get out of this infinite loop to swap the
     partition element into place}
    if (i >= j) then Break;
    {otherwise swap the two out-of-place elements}
    Temp := aItemArray[i];
    aItemArray[i] := aItemArray[j];
    aItemArray[j] := Temp;
    {and continue}
    inc(i);
    dec(j);
  end;
  {swap the partition element into place, return the dividing index}
  aItemArray[R] := aItemArray[i];
  aItemArray[i] := Last;
  Result := i;
end;
```

➤ *Listing 8: The Partition routine for quicksort*

ways around some of these points in a moment.

Because it is recursive we have to consider the stack size required by the routine. If we're lucky and don't hit a 'bad' set that causes quicksort to deteriorate, although we'll be discussing ways of getting round this, we split the set into two at each pass and call the quicksort routine recursively, once for each set. So, the maximum number of times we'll recurse is log *n* where *n* is the number of elements in the set. So for 1,000,000 items we'll go to about 20 levels. This is well within the ability of the 32-bit stack. If we do hit a bad set, we'll recurse about *n* times, not very good at all.

If we assume there exists a routine called Partition, then quicksort can be coded recursively as in Listing 7. The first line of code is the recursion terminator: there's no need to do anything further if the number of elements to sort is less than or equal to 1, by definition a set of one element (or less!) is already sorted. If this condition does not apply then we partition the set, and then quicksort the two subsets created by the partition by recursively calling the `QuickSortPrim` routine.

The Partition routine is a little more involved and is shown in Listing 8. We first set the left and right indexes, i and j, and save the partition element (using a local variable will be faster than accessing the array element in the loops that follow). We now enter an infinite loop and will break out of it when the indexes cross. We could use a normal while loop at this point, checking for index j crossing i, but the code inside the loop would have the same check as well, making two checks per loop instead of one. The first thing to do in the loop is to find the first element from the left that is greater than or equal to our partition element. Note that we know we won't fall off the end of the array with this loop since our partition element acts as a sentinel. Next, we find the first element from the right that is less than our partition element. Here we *do* have to explicitly check for falling off the end of the array: the partition element may be the smallest item in the set. If the indexes have crossed we know we've found the correct position for the partition element and so we break out of the infinite loop. Otherwise the elements found are out of place and are swapped over. Round the loop we go again. Once we've broken out of the loop we swap the partition element into place and return the index where it divides the set.

Making The Right Move

One thing I mentioned early on, and one that you would have noticed if you'd tried a quicksort with cards, is that once the sets get very small there's a lot of trivial work going on to partition and sort them. If a subset has three or four elements, for example, the partitioning exercise seems to take forever and sorts the elements in a quite remarkably roundabout way. One speed improvement we can do to quicksort is to stop recursively sorting subsets, once the subset is less than some number of elements. Think about what the overall set looks like at that point. All the elements in the set are roughly in the right place, but they may be adrift by at most our cut-off number of elements. I took my example deal of Hearts I used above, set the cut-off at 3 elements (ie, subsets with 3 or less cards were not quicksorted) and obtained the following series:

```
J 1 5 2 Q K 9 10 7 4 8 3 6
3 1 5 2 Q K 9 10 7 4 8 J 6
3 1 5 2 4 K 9 10 7 Q 8 J 6
3 1 5 2 4 6 9 10 7 Q 8 J K
3 1 2 5 4 6 9 10 7 Q 8 J K
3 1 2 4 5 6 9 10 7 Q 8 J K
3 1 2 4 5 6 9 10 7 8 Q J K
3 1 2 4 5 6 9 10 7 8 J Q K
3 1 2 4 5 6 7 10 9 8 J Q K
3 1 2 4 5 6 7 8 9 10 J Q K
```

Notice that we have four elements fixed in place, and the remaining nine are either fortuitously in the correct position, or *very* close. So what? I hear you ask. Well, if we did an insertion sort on the partially sorted subsets when we get to them, it would be very fast, since elements would not have to move far at all.

So a good improvement to quicksort is as follows. First, set a cut-off point. Second, subsets that have this number of elements or less are not quicksorted; we use an insertion sort instead. Brilliant! But the question then arises: what is the cut-off point? Three, like my card example? Or larger? Well, the card example is somewhat of a forced case: we're only sorting 13 cards after all, and I selected a cut-off of three just to illustrate the point. Research has shown that there is no ideal cut-off: any value from 5 to about 20 is good. It depends too much on the machine the sort is running on, how the compiler converts the code into machine code, the size of the elements, the number of elements. Using 10 is as good as we'll get without fine-tuning the quicksort for our particular environment.

We now need to think about the partition element, the element about which we divide up a set. Ideally, we would like to choose the partition element that splits the set into roughly two equal halves. In the algorithm we have used so far, we've just chosen the rightmost element without worrying about whether this is a good choice. (Indeed I showed that, in one particular case - a fully sorted set, it's the worst element to choose.) Unfortunately, without scanning the entire set, there is no way to find out the exact median element, except by chance (the median element of a set would split the set into two equal halves, and indeed the most efficient way to find it is by a partitioning algorithm). If the set of elements was in a random order, our simple strategy would be pretty good, *on the average*. Sometimes it would be close to the actual median, sometimes way off. We can do better, though.

One simple strategy for determining a partition element is to use a random selection from the set. If the set has *n* elements, then generate a random number index between 0 and *n*-1 and use that element as the partitioning element, swap it over with the final element and then proceed with the normal partitioning code. This algorithm relies on probability (and your random number generator!) to improve the running time of quicksort (the probability of always selecting the worst partition element is vanishingly small).

Another, but probably more effective, strategy to choose the partition element is the Median Of Three method. Take the first element, the last element and the one in the middle. Find the median of these three elements (ie, sort them, and use the middle element) and swap it with the last element. Now we proceed to partition the set as before. What we've done with this strategy is to take a small sample of the elements in the set and assumed that the median of this sample is roughly the same as the median of the entire set. For a randomly ordered set, this strategy is much better than the one we have been using. We're getting a better guess at the partition element, and generally it'll be closer to the median than just taking a single element. For an already sorted set, this strategy is brilliant: we'll always choose the correct median element and the partitioning process will be pretty well optimal.

And there's another great thing that the Median Of Three method gives us for free if we're prepared to do a little extra work. Swap the middle element of the set with the last but one element. Sort the first, penultimate and last elements, so that the smallest is at the first position, the median element at the last but one position and the largest at the final place. Now we can perform the partition on the set formed by the elements between the second and penultimate elements (the first and last are already correctly partitioned). The first element of the set forms a sentinel for the partitioning process on the left hand side, and we can get rid of the extra check in an inner loop of the Partition routine.

The final quicksort and partition code can be found on this month's diskette. It includes the two improvements we have discussed: the Median Of Three strategy, and the cut-off point and insertion sort. Compared with the original quicksort, the improved version is about 25% faster for sorting a million doubles (the percentage decreases for smaller sets, for 100,000 doubles it's about 16% faster).

Mama Didn't Raise No Fool

Well, we've reached the end of the road for this particular article. We've seen six different sort algorithms: bubble sort, shaker sort, selection sort, insertion sort, Shellsort, and quicksort. Two others were briefly mentioned as being prime targets for further articles: merge sort and heap sort. We briefly discussed the pros and cons of these six and showed some improvements. We managed to whittle them down to a top two: improved quicksort for the general case, and selection sort for the case where moving an element is expensive in time.

I would also put forward two other sorts for extremely specific cases. The first is insertion sort, for the case where not many elements are out of order (I once used an insertion sort where I wanted to add a single element to an already sorted list: I added it to the end and then insertion sorted the list). The second is Shellsort, for the case when you have to code the sort yourself. The quicksort code is extremely easy to get wrong (or, conversely, hard to get right) and you'll never notice until it's too late. When I was writing the code for this article, the only sort I had problems coding was the quicksort, all the others worked first time. Indeed, I almost (but only just!) shipped our esteemed editor a completely broken quicksort in the source files that accompany this article: I just didn't notice until I got worried having noticed that my quicksort was slower than my Shellsort. Not only slower but it would produce a sort with items out of place. *Caveat programmer.*

Anyway, to illustrate the sorts, I've included on the disk a program that shows them in action. You can run each sort on a set of items and the program shows the moves being made to sort the items by displaying them as lines and moving the lines. This program gives you a good feel for how the individual sorts move items around to get them in the correct place; it does not, however, attempt to show you any performance characteristics. Indeed, the sort that fares the best with this program is the selection sort: because the time taken to display the lines when swapping elements is much larger than comparing elements, selection sort outdoes all the others. Also, the program demonstrates very well how shaker sort got its name.

Julian Bucknall is a sort of programmer. He does have a sister, but she doesn't swing out that often. The code that accompanies this article is freeware and can be used as-is in your own applications.