1. Bubble Sort

Bubble sort is one of the simplest sorting algorithms available, and perhaps as a consequence one of the worst.

The name gives some indication of how the technique works - each item 'bubbles' it's way up the list into it's proper place. Bigger bubbles move further up the list, leaving the smaller ones in their wake. More specifically, the process is as follows, for a list with N items:

1. Set C to N-1.
2. For items 1 through C, compare each with the item after it, and swap if not in the correct order.
3. Decrement C by 1.
4. If C is ≥1 and at least one swap was performed on the most recent pass, go back to step 2.

In summary, each complete pass through the list has the effect of pushing each value slightly closer to it's sorted position. After a finite number of passes (at most N-1) the list must eventually become completely sorted.

Example

Consider this example list, below. Let's use bubble sort to order it in ascending order.

4 8 1 9 3

First we would compare 4 and 8. They are in the correct order. So we move to the next pair, 8 and 1. They are not in the correct order, so they are swapped. The list now looks like:

4 1 8 9 3

Now we compare 8 and 9. They are in the correct order, so we move on to compare 9 and 3, which are not correctly ordered, and so are swapped. The list now looks like like:

4 1 8 3 9

That concludes one single pass. We now start again from the beginning, for the second pass. 4 and 1 are out of order, so they are swapped, and the list becomes:

1 4 8 3 9

4 and 8 are already correct, so they are left as is. 8 and 3 however need to be swapped, so the list becomes:

1 4 3 8 9

Note that we don't bother comparing 8 and 9 on the second pass, because we know they are already in the right order. Thus we now move straight to the next pass. By comparing 1 and 4, we find that they are correctly ordered. But 4 and 3, the next pair, are not and so are swapped. This results in the list:

1 3 4 8 9

It can be seen quite obviously that this small example is now sorted. However, the bubble sort algorithm presented would still perform the next pass, obviously to no effect.

Performance

The performance of bubble sort can be analysed quite simply, because it is a very simple algorithm. On the first pass, you perform N-1 comparisons. On the second, if a second is necessary, you perform N-2 comparisons. Etc. Some simply integration will reveal that the average number of comparisons, per pass, is in fact N/2. Given that the average number of passes is (N-1)/2, this ends up giving an average number of comparisons for a list of N items as:

$$\frac{3}{8}N^2 - \frac{3}{8}N$$

Working out how many swaps you do is a much lengthier mathematical proof, so for the sake of brevity I won't include it. Just take my word that it turns out you do, on average, one swap for every two comparisons - this is somewhat intuitive anyway. Consequently, the average number of comparisons for a list of N items is:

$$\frac{3}{16}N^2 - \frac{3}{16}N$$

That's the average case. The other two cases of interest are the best and worst. In the best case, when the list is already sorted, the first pass results in no swaps, so the sort finishes. Thus there are just N-1 comparisons.

In the worst case, when the list is in reverse order, you have to perform all N-1 passes, and you end up having to perform a swap for every comparison. This means the number of swaps performed is roughly:

$$\frac{1}{2}N^2 - \frac{1}{2}N$$

When you compare these cases to other sorting algorithms, bubble sort looks pretty abysmal. It is on average an O(N^2) algorithm, meaning that for every doubling of list size, the sort takes about 4 times longer. Thus for very large lists, bubble sort becomes incredibly slow, unless it happens to be close to it's best case.

Electoral Sort Pro's & Con's

So what are the specific pro's and con's of bubble sort in terms of the electoral sort problem? Well, the short of it is that bubble sort is abysmal for this problem. The list is already sorted by family name, but postcode is in no particular way related to family name, so the postcodes are essentially in completely random order. Thus the average case applies, which means O(N^2) comparisons and swaps.

There are no significant advantages for bubble sort, over the other algorithms we've investigated. In terms of speed, bubble sort only compares more favourably to other algorithms in it's best case, when the list is pretty much already sorted. In our problem, this is simply not true.

One advantage bubble sort does have over some other algorithms is that it is in-place. That is, it performs all it's operations on the original list, without allocating any temporary storage or sublists or whatever else. This means it uses the minimum amount of memory necessary to sort the list. On a computer with little memory available, bubble sort's relative performance can be significantly better, compared to out-of-place algorithms like merge or radix sort. That is not to say, however, that it won't ultimately still be much worse.

2. Radix Sort

Radix sort is one of the best general sorting techniques known. The general technique is very simple, being based on common human behaviour. When presented with a large pile of library cards, for example, most people will start to sort them by first dividing them into piles based on the first letter of their title. They then sort each of the subpiles using the second character of the title, and so forth. Once all the piles at all levels have been sorted, it's a simple matter to concatenate them all back together into the final sorted list.

Formally, the procedure is:

5. Allocate B sublists, where B is the number of unique values the index may have
6. For each item in the list, move it into the appropriate sublist based on the current index field
7. Repeat this procedure for each non-empty sublist, but with the next field as the index
8. Append the contents of the first (by order) sublist order to the original list. Then append the contents of the second (by order) sublist. Proceed through to the last sublist.

So it can be seen that radix sort is somewhat similar in principle to merge or quick sort, except that instead of having two sublists at each level, there are usually many (B is often between 10 and several hundred). In computing terms, it then becomes very fast to sort each item, because you need only index into an array using some simple key, e.g. a single character of a larger string. Even more importantly, radix is what's known as a stable sorting algorithm (explained later), which in our particular case allows it to be optimised to perform extremely well.

Example

Take the list of numbers below. We'll consider each one to be composed of 3 fields, their 3 digits, and will start with the most significant digit.

752, 257, 326, 226, 731, 398, 757

The first step is to sort these numbers into sublists based on their first digit. Thus we end up with 3 subgroups - one each for the digits 2, 3 & 7 - as shown below.

2: 257, 226
3: 326, 398
7: 752, 731, 757

We now sort each subgroup into further subgroups, based this time on the second digit. So the 2: group above gets broken down into a further 2 subgroups, as shown below. Likewise, the 3: group above gets broken down into 2 subgroups. And finally the 7: group above gets broken down into 2 subgroups.

```
2:    2: 226
      5: 257
3:    2: 326
      9: 398
7:    3: 731
      5: 752, 757
```

The last step is to break down that last pair of numbers in the 7:5: group above, based on the third digit. The final result is a series of groups and subgroups that look like the following:

```
2:    2: 226
      5: 257
3:    2: 326
      9: 398
7:    3: 731
      5:    2: 752
            7: 757
```

If we now put these back into a single list in descending numerical group order, we get the final sorted list shown below.

226, 257, 326, 298, 731, 752, 757

Performance

Radix sort is very hard to analyse for many reasons, not the least of which is because in practice it can be heavily optimised for many specific cases. It can also be implemented many different ways even for the same task. So it's hard to say exactly what it's best, average and worst cases are. As such, I won't bother with a general case, suffice to say it usually performs at least as fast as quicksort's average case, and reliably so. I'll go into the details of the performance with regards to our use of it in the electoral sort problem, in the Pro's & Con's section, later. First, the terms 'stable' and 'unstable' need to be defined as they relate to sorting algorithms.

Stable vs Unstable Sorting

A stable sorting algorithm is one which doesn't change the order of 'identical' items. This may sound a bit silly in the context of simple integer lists - where you can't tell identical items from each other - but when you start to deal with more complex data, it can be very important. For instance, if we were sorting a list of people by their surnames alone, then there may well be 'identical' items - two or more people with the same surname. But these 'identical' people are obviously not really identical - they may of course have different first names, ages, residencies, etc.

There are many practical reasons why you might want a stable sort. For example, consider a scenario where you are re-sorting already sorted data. For instance, if the list of names previously mentioned was already sorted based on first name, but you wanted it instead based on family name and then first name. There are two approaches - with an unstable sort, you would have to re-sort each item based on family name, and then again on first name within each family. But with a stable sort, you could re-sort only based on the family name, because you know for certain that within each family name, the first name's are still in order.

Consider a very simple example, with the list of names below:

> Albert Whit, Brian Smith, Cedrick Leacle, Darren Smith, Edgar Linch

This list is already sorted by first name. To re-sort it by family name using an unstable sorting algorithm, you may (or may not; unstable algorithms are usually unpredictable) end up with the following:

> Edgar Linch, Cedrick Leacle, Darren Smith, Brian Smith, Albert Whit

Note that our two Smith's are not in the correct order. Somewhere in the process of sorting by family name, they were swapped around. So we'd have to now perform a subsort on all the Smith's to put them back into the right order.

Alternatively, we could just use a stable sort to start with, which guarantees that after sorting by family name, the first names will still be in exactly the same order as before. No need to waste time re-sorting on first name. Thus, the direct result is:

> Edgar Linch, Cedrick Leacle, Brian Smith, Darren Smith, Albert Whit

Most sorting algorithms can be implemented as stable - e.g. bubble, radix, quick, merge, selection, etc. However, some end up performing rather poorly when implemented as stable, most notably quicksort.

Electoral Sort Pro's & Con's

Radix sort is, overall, extremely good for the electoral role problem. It will inevitably be the fastest algorithm, because it's stability means the electoral role need only be re-sorted based on postcode, not all the other data. And since it need only use the postcode as an index into a list of sublists, there are no real comparisons, swaps, or other operations to be performed. Appends to a vector are on average quite quick, certainly not much worse than a swap, and much better than an insert or delete. So overall radix sort ends up being much faster.

There is however a potential disadvantage with radix sort, which is that it is an out-of-place algorithm. This means it has to create temporary data structures to perform the sort, in addition to the initial list of items. This can be a huge problem for larger lists, as you need proportionately more and more memory. If swapping has to occur, the performance can drop by several orders of magnitude. Thus, for very large lists and/or limited memory, radix sort probably won't perform as well as some of the in-place algorithms, e.g. quicksort.

3. Predicted Performance

Selection sort

Selection sort requires, in total, N-1 passes, the first with N-1 comparisons, the second with N-2, etc. This is the same as for bubble sort, so the average number of comparisons is, like bubble sort, approximately:

$$\frac{1}{2}N^2 - \frac{1}{2}N$$

Unlike bubble sort, however, this is also the best and worst case - selection sort never knows whether the list is already sorted or not, so it must perform all N-1 passes. However, it does compare more favourably in terms of the number of swaps. Each pass requires at most a single swap. So there are at most N-1 swaps in total, the best case being 0, the worst N-1, and the average - while very dependent on the data set - generally approaching N-1.

Insertion Sort

This requires N-1 items to be inserted at an unknown position in the sorted part of the list. For the first item, there is just one comparison, and at most one insert. For the second, there is either one or two comparisons, and at most one insert. So the average number of comparisons per item works out to be approximately N/4 (about C/2 comparisons for each item C, where C is 1 to N-1 and so averaging about N/2), and the average number of inserts approaches N-1 (for the whole list). Thus, the average number of comparisons is:

$$\frac{1}{4}N^2 - \frac{1}{4}N$$

However, in the best case there are just N-1 comparisons (each item is compared with the item before or after it, and found to be in the correct order) and no inserts. Thus insertion sort can be very very fast for a sorted list. In the worst case, a reverse ordered list, each item requires twice as many comparisons as average:

$$\frac{1}{2}N^2 - \frac{1}{2}N$$

And each item requires an insert, so N-1 inserts. How expensive an insert & delete is determines how well insertion sort measures up overall - generally an insert or delete is very expensive compared to a swap, often increasing in cost with the size of the list.

Mergesort

While merge sort can be done in place with just swaps, I have implemented it out-of-place with vector appends, so I'll analyse the latter case only.

Each level requires N comparisons in total (first level requires N, second level requires 2 times N/2, etc) when merging the two sublists together. There are at most ln(N) levels. Consequently, it requires at most Nln(N) comparisons. For every comparison, the result is that one of the compared items is appended, so there are also Nln(N) appends. This is it's best, average and worst case.

Quicksort

Quicksort is terribly difficult to study mathematically. Apart from the fact that it's speed varies considerably based on the data it is sorting, there are several ways of selecting the pivot, which can greatly effect performance. It is however very similar to merge sort, so a lot of the basic conclusions apply - there are ln(N) levels, and each one requires about N comparisons, therefore the number of comparisons is of the order of Nln(N). It can in fact be shown empirically that quicksort has an average case - assuming a random pivot - of approximately 1.39Nln(N) + O(N) comparisons, and half that many swaps. In the best case, there are significantly fewer swaps, while in the worst there are much more. How few or many is difficult to decide upon. It can be shown that for an already sorted list, and selecting the first or last item as the pivot, the number of comparisons and swaps both approach something on the order of N^2, making worst case quicksort about as a efficient as bubble sort.

The Algorithms Compared

	Comparisons			Swaps/Moves/Appends		
	Best	Average	Worst	Best	Average	Worst
Insertion	N-1			0 M	-> N-1 M	N-1 M
Selection				0 S	-> N-1 S	N-1 S
Quick	Nln(N) + O(N)	1.39Nln(N) + O(N)	O(N^2)	0 S	0.69Nln(N) + O(N) S	O(N^2) S
Merge	Nln(N)	Nln(N)	Nln(N)	Nln(N) A	Nln(N) A	Nln(N) A

So, which algorithm is the best? It's a difficult question to answer in general, but in terms of the electoral sort problem, it can be guessed at with reasonable confidence.

Presumptions

First, let us presume that the cost of an append will not be significantly larger than a swap. Intelligent use of the reserve() method could improve the speed, but on the other hand the extra memory required by an out-of-place sort acts against it anyway. Consequently, I'll presume an append takes about 3/2 times longer than a swap, to try and take into account all these things.

The cost of a move, on the other hand, is very high, and proportional to the size of the list. Even for a small list, it is likely to be quite big - at least an order of magnitude more than a swap. For very large lists, it will only get worse.

The final unspoken presumption I've made is of the hardware that will be running the sort. The intended hardware in my case is an 800mhz G3 750CX (512k L2 @ 800mhz) with 256 mebibytes of 100mhz SDRAM, a 4200rpm UDMA hard drive, running MacOS 10.2.6 and using gcc 3.1. This machine is obviously clearly lacking in memory and disk bandwidth, but does provide reasonable integer performance. So it ultimately favours in-place sorts, and performs comparisons relatively cheaply as compared to other platforms (for reasons which are beyond the scope of this report to explain).

Discussion

It's immediately obvious that as the list size increases, insertion sort will become less and less attractive. So while it's average number of comparisons is half that of selection sort, this advantage will eventually be outweighed by the increasing cost of moves. But for the size of the files we're using, up to 200000 entries, a list of pointers only takes up 100 kibibytes. Given that modern computers can in practice move at least 3 gibibytes per second (allowing for L2 cache), this theoretically still allows for up to 20000 moves per second. Thus, I predict insertion will still be significantly faster than selection, even for these sizes.

Whatever the small difference between insertion and selection, it's clear that quick and merge will both be orders of magnitude faster. We can assume our data is a reasonably average case, so quick sort should perform close to it's average case, and thus quite well.

Even though quick sort requires slightly more comparisons than merge, it requires significantly fewer swaps than merge's appends. Given swaps are slightly faster than appends anyway, I predict quick sort will be slightly faster than merge - maybe twice as fast, but no more. The weighting factor is the fact that my merge sort implementation is out-of-place, versus quicksort's in-place, so the extra memory footprint will be a significant burden for mergesort.

So, in conclusion, I predict the following rankings: Quicksort, followed reasonably closely by merge, and then far behind insertion, with selection relatively close behind that. I should also mention that, although not analysed here, I expect radix to be faster than even quicksort, and bubble to be much slower than even selection sort.

4. Analysis

The test was performed in a cut-down single-user mode, without any system services or unnecessary processes running, including the virtual memory system, network stack and other niceties. Average free memory prior to running each test was 210 mebibytes. File caching was apparently off, although the test results do not include data transfer to and from disk anyway. All test outputs were compared with "diff --brief" and found to be byte-identical. Any one of the results for each role file were inspected by hand to ensure they are at least correct at a glance.

I used my own timing mechanisms because I feel the inclusion of loading and saving times would pollute the timing data. The mechanism I ended up using was a simple system time difference. This is not the process time used by the "time" command, among others. On my system the 'clock' the "time" command uses has a resolution of 10ms. As can be seen, many of the sorts were significantly faster than this, especially for the smaller files. Consequently the results would be meaningless using "time". For verification, I did observe the "time" taken for each test, to ensure the two timing measurements did not differ significantly. None of them did, because as mentioned only a few other processes were running - kernel_task, init, mach_init and sh. Over the hour I ran the tests, these processes used collectively about 10 seconds.

The raw results are tabulated below, and a semi-logarithmic graph of the execution time is attached. As can be seen, there were really only two main classes of performance. Bubble, selection and insertion fell into the slower class, and showed their O(N^2) nature. The other 4 sorts show a much more linear loss of performance, more consistent with their general O(Nln(N)) nature.

I included AVL sort mainly because I could - the code came easily from one of our labs - and because it initially seemed to outperform all other methods. After much tweaking to the other sorting methods, however, it's advantage was lost, and the final results can be seen below.

Time Taken In Seconds

	AVL	Quick	Radix	Merge	Insertion	Selection	Bubble
20	0.000895977	0.023625	0.00201201	0.000890017	0.000850916	0.000862956	0.00079596
200	0.00135601	0.000982046	0.00211501	0.001472	0.00168395	0.00216103	0.00217295
2000	0.00734794	0.00386393	0.00358999	0.00884807	0.080317	0.141952	0.175975
20000	0.129855	0.063136	0.014948	0.113355	17.9085	34.0334	88.1006
200000	3.01827	1.08507	0.11332	1.60242	<timeout>	<timeout>	<timeout>

Comparisons/Swaps/Moves/Appends

	AVL	Quick	Radix	Merge	Insertion	Selection	Bubble
20	61C, 20A	114C, 21S	40M	65CA	111C, 18M	190C, 17S	190C, 97S
200	1267C, 200A	2322C, 331S	400M	1290CA	10619C, 195M	19900C, 193S	19900C, 10425S
2000	19501C, 2000A	33466C, 4953S	4000M	19375CA	1000554C, 1983M	1999000C, 1993S	199900C, 998563S
20000	268332C, 20000A	464779C, 64891S	40000M	260794CA	99493756C, 19983M	199990000C, 19993S	1999900C, 99473761S
200000	3401391C, 200000A	5793076C, 802984S	400000M	3272975CA	<timeout>	<timeout>	<timeout>

I'll point-form the significant conclusions from this data, to save a long and rambling discussion:

- Strangely, quicksort always performed very poorly for the 20 record file. I can only presume this happened to be a very bad case. Even so, it's consistent sluggishness for this one case is peculiar.
- AVL tree sort, added in because the source was available in one of our labs, shows reasonably good performance, but does start to slow down non-linearly with larger and larger files

· The specially optimised radix sort is the clear winner, outperforming every other sort on the 2000 record or larger files.
· Radix, merge and quicksort all perform relatively poorly for the 20 and 200 record files, as expected due to their overheads. It's clear that for around 20 records or less, insertion, selection and even bubble sort outperform them. Results like these lead to the inevitable 'combo' or 'quicker' sort algorithms, utilising quicksort for large data sets, and insertion sort for 25 records or smaller.
· Selection sort is getting slower at a greater rate than insertion, despite my suspicions that insertion would deteriorate faster. Clearly the cost of inserts was not as large as I predicted, at least for 20000 records or less.
· Selection sort was given half an hour on the 200000 record file, and did not complete. It seems safe to assume that insertion and bubble would also be of the same order - that of hours, rather than seconds.
· Radix sort has the highest initial overhead of all the algorithms, probably due to having to create 10000 vectors outright. But by about 2000 records or so it begins to outperform even quicksort.
· Merge sort began to approach quicksort's performance for very large lists. It seems probable that a 2000000 record file would see them perform roughly equivalent, extrapolating from the graph.
· Insertion, selection and bubble sort are actually showing slightly worse than O(N^2) performance.
· Likewise, quicksort and AVL sort are both showing slightly worse than O(Nln(N)) performance. They appear to be degrading at a similar rate.
· Merge sort is showing very close to O(Nln(N)) performance.
· Radix sort is showing better than O(Nln(N)) performance.
· Merge sort does indeed perform significantly fewer comparisons than quick sort, but as expected it's more-expensive appends keep it from being faster. Plus the fact that it's always performing about 3 times more appends than quick is doing swaps. In fact, this almost indicates that appends are faster than swaps.
· Even though selection and insertion perform far more comparisons than bubble sort, they perform far far fewer swaps and insertions, thus the performance advantage

Environmental Considerations

My reduced testing environment is obviously not a good example of a real world scenario. However, my test machine is not a good example to start with - 256 mebibytes of memory is very minimal, as is the speed of the 4200rpm hard drive. In the end I felt that choosing any particular level of free memory was an arbitrary choice anyway, and would thus produce non-general results. So, I aspired to ensure the environment had more than enough for every test, thus ensuring fairer results.

Another condition to note was the size of the executable - 270k (c++stdlib and STL statically linked, compiled with -Os and stripped). Given 32k L1 and 512k L2 caches, this would leave at best around 250k of L2 cache available for data. Thus I would have preferred to implement each sort in it's own executable, to not only improve performance in general, but to subtly include the performance penalty due to more complicated code. The AVL tree, for example, included a lot of inlinable code which would have resulted in a relatively large executable. The core bubble sort algorithm, on the other hand, was probably tiny, even with aggressive inlining.

Significant Optimisations

General

· Use of 'register' keyword in key places improved performance when compiling without optimisations
· Use of static inline functions improved performance significantly in some places
· Removed use of endl throughout program - improved output speed to file by 300%
· Use of reserve() method of vector class where possible, improving performance slightly

Bubble sort

· Removed the check for whether any swaps had been performed on the last pass, because in most cases you perform close to N-1 passes anyway, and the check itself was slowing things down

Radix sort

· Sort into 10000 buckets, indexed 0 through 9999, rather than 4 levels of 10 buckets, because this reduces total memory usage, number of appends, and consequently improves outright speed significantly

Bibliography

I should note that every algorithm, apart from the AVL tree code, was written from scratch by myself. In each case, I learnt how the algorithm was supposed to work from English or pseudo-code descriptions, and then implemented it as best I could. I have not at any point compared my code with any other. I'm hoping this assignment is a measure of how well I can implement each of the sorts, not how well some random guy and his website can.

My primary sources of information were:

Sorting Algorithms <http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html>
James Gosling, Alvin Raj, Jim Boritz and Jason Harrison

Sorting Algorithms <http://cs.smith.edu/~thiebaut/java/sort/demo.html>
Biliana Kaneva and Dominique Thiébaut

Animation of Sorting Algorithms <http://math.baruch.cuny.edu/~bshaw/index-8.html>
B.I. Shaw

Sorting Algorithms <http://linux.wku.edu/~lamonml/algor/sort/>
Michael Lamont

Sorting and Searching Algorithms: A Cookbook
<http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/niemann/s_man.htm>
Thomas Niemann

Introduction to Sorting Laboratory <http://www.ship.edu/~cawell/Sorting/main.htm>
Carol A. Wellington

Sorting Algorithms <http://www.cs.hope.edu/csci120/topics/sorting.html>
Dr. McFall

Sorting Algorithms <http://www.cobolreport.com/columnists/leif/07242000.htm>
Leif Svalgaard