Comparison of several sorting algorithms

Introduction

From time to time people ask the ageless question: Which sorting algorithm is the fastest? This question doesn't have an easy or unambiguous answer, however. The speed of sorting can depend quite heavily on the environment where the sorting is done, the type of items that are sorted and the distribution of these items.

For example, sorting a database which is so big that cannot fit into memory all at once is quite different from sorting an array of 100 integers. Not only will the implementation of the algorithm be quite different, naturally, but it may even be that the same algorithm which is fast in one case is slow in the other. Also sorting an array may be different from sorting a linked list, for example.

In this study I will only concentrate on sorting items in an array in memory using comparison sorting (because that's the only sorting method that can be easily implemented for any item type, as long as they can be compared with the less-than operator).

The testing environment

In order to test the speed of the different sorting algorithms I made a C++ program which runs each algorithm several times for randomly-generated arrays.

The test was run in a Pentium4 3.4GHz running Suse 9.3, and the program was compiled using gcc 3.3.5 with the compiler options "`-O3 -march=pentium4`".

The `drand48()` function of glibc was used for random number generation. This should be a rather high-quality random number generator.

Four different array sizes were used: 100, 5000, 100000 and 1 million items (the last one used only on the integer array tests). Random numbers between 0 and 10 times the array size were generated to create the array contents, except for the high-repetition test, in which numbers between 0 and 1/100 times the array size were generated (which means that each item is repeated in average 100 times).

Four different random number distributions were used:

1. Completely random.
2. Almost sorted: 90% of the items are in increasing order, but 10% of randomly-chosen items are random.
3. Almost reversed: Like above, but the sorted items are in reverse order.
4. The array is already sorted, except for the last 256 items which are random. (This case was used to test which sorting algorithm would be best for this kind of data

container, where items are kept sorted and new items are added to the end, and the entire container sorted after the amount of items at the end grows too large.)

Four different testcases were run:

1. Items are 32-bit integers. These are both very fast to compare and copy.
2. Also 32-bit integers, but with a high number of repetitions. Each value in the array repeats approximately 100 times in average.
3. Items are C++ strings with identical beginnings. Strings of 50 characters (with only the last 8 characters differing) were used for the test. This tests the case where copying is fast but comparison is slow (copying is fast because the strings in gcc use copy-on-write).
4. Items are arrays of integers. Arrays of 50 integers (ie. 200 bytes) were used. Only the first integer was used for the comparison. This tests the case where comparison is fast but copying is slow.

More detailed info for these testcases is given in their individual pages.

I tried to implement the program so that it first counts how much time is spent generating the data to be sorted, and then this time is substracted from the total time (before dividing it by the number of loops). While it's not possible to do this in a very exact way, I'm confident that the results are close enough to reality.

Each testcase with each sorting algorithm was run several times, every time with different random data (about 100-10000 times depending on the size of the array). This was done to average out individual worst cases.

The sorting algorithms

Insertion sort

Insertion sort is good only for sorting small arrays (usually less than 100 items). In fact, the smaller the array, the faster insertion sort is compared to any other sorting algorithm. However, being an $O(n^2)$ algorithm, it becomes very slow very quick when the size of the array increases. It was used in the tests with arrays of size 100.

Implementation.

Shell sort

Shell sort is a rather curious algorithm, quite different from other fast sorting algorithms. It's actually so different that it even isn't an $O(n\log n)$ algorithm like the others, but instead it's something between $O(n\log^2 n)$ and $O(n^{1.5})$ depending on implementation details. Given that it's an *in-place* non-recursive algorithm and it compares very well to the other algorithms, shell sort is a very good alternative to consider.

Implementation.

Heap sort

Heap sort is the other (and by far the most popular) *in-place* non-recursive sorting algorithm used in this test. Heap sort is not the fastest possible in all (nor in most) cases, but it's the *de-facto* sorting algorithm when one wants to make *sure* that the sorting will not take longer than O($n$log$n$). Heap sort is generally appreciated because it is trustworthy: There aren't any "pathological" cases which would cause it to be unacceptably slow. Besides that, sorting in-place and in a non-recursive way also makes sure that it will not take extra memory, which is often a nice feature.

One interesting thing I wanted to test in this project was whether heap sort was considerably faster or slower than shell sort when sorting arrays.

Implementation.

Merge sort

The virtue of merge sort is that it's a truely O($n$log$n$) algorithm (like heap sort) and that it's stable (iow. it doesn't change the order of equal items like eg. heap sort often does). Its main problem is that it requires a second array with the same size as the array to be sorted, thus doubling the memory requirements.

In this test I helped merge sort a bit by giving it the second array as parameter so that it wouldn't have to allocate and deallocate it each time it was called (which would have probably slowed it down somewhat, especially with arrays of the bigger items). Also, instead of doing the basic "merge to the second array, copy the second array to the main array" procedure like the basic algorithm description suggests, I simply merged from one array to the other alternatively (and in the end copied the final result to the main array only if it was necessary).

Implementation.

Quicksort

Quicksort is the most popular sorting algorithm. Its virtue is that it sorts *in-place* (even though it's a recursive algorithm) and that it usually is very fast. On reason for its speed is that its inner loop is very short and can be optimized very well.

The main problem with quicksort is that it's not trustworthy: Its worse-case scenario is O(n^2) (in the worst case it's as slow, if not even a bit slower than insertion sort) and the pathological cases tend to appear too unexpectedly and without warning, even if an optimized version of quicksort is used (as I noticed myself in this project).

I used two versions of quicksort in this project: A plain vanilla quicksort, implemented as the most basic algorithm descriptions tell, and an optimized version (called `MedianHybridQuickSort` in the source code below). The optimized version chooses the median of the first, last and middle items of the partition as its pivot, and it stops partitioning when the partition size is less than 16. In the end it runs insertion sort to

finish the job.

The optimized version of quicksort is called "Quick2" in the bar charts.

The fourth number distribution (array is sorted, except for the last 256 items which are random) is very expectedly a pathological case for the vanilla quicksort and thus was skipped with the larger arrays. Very unexpectedly this distribution was pathological for the optimized quicksort implementation too, with larger arrays, and thus I also skipped it in the worst cases (because else it would have affected negatively the scale of the bar charts). I don't have any explanation of why this happened.

Implementation.

std::sort

I also used the C++ standard sorting function (as implemented in gcc) for comparison. This is a highly-developed sorting function which is somewhat similar to the optimized quicksort I described above, but with further optimizations (such as reverting to heapsort if the sorting seems to be too slow).

How to read the comparisons/assignments tables

Besides a bar chart, each testcase includes a table of numbers. This table presents the amount of comparison and assignment operations performed by the sorting algorithm *in average* during the sorting of one input array.

The amount of comparisons and assignments and their relative ratio is not extremely relevant when comparisons and assignments are fast (such as with integers). However, when comparison or assignment of the elements is a heavy operation then it may be much more interesting and relevant to compare different sorting algorithms with regard to how much they perform those operations compared to the other algorithms.

Each table has the following format:

	Random		**Almost sortd**		**Almost rev.**		**Random end**	
Alg.	**Comp.**	**Assig.**	**Comp.**	**Assig.**	**Comp.**	**Assig.**	**Comp.**	**Assig.**
(name)	(value)	(value)	(value)	(value)	(value)	(value)	(value)	(value)
...	...	...	...	...	...	...	...	...

The first column specifies the sorting algorithm. Each row has pairs of numbers, one pair for each input data distribution type. They represent the amount of comparisons and assignments the algorithm needed (in average) to sort the array once.

The results

1. Integers

2. Integers, high amount of repetition
3. Strings: Slow comparison, fast copying.
4. Arrays: Fast comparison, slow copying.

Conclusion