Sorting revisited.

I previously posted an algorithm that allows one to sort with less than all the facts (specifically, if computing the transitivity of "<" is not trivial.) But here I'd like to revisit just straight sorting.

Conventional wisdom is as follows:

- quicksort is the fastest sorting algorithm in practice but has a number of pathological cases that can make it perform as badly as **O(n^2)**.
- heapsort is guaranteed to run in **O(n*ln(n))** and requires only finite additional storage. But there are many citations of real world tests which show that heapsort is significantly slower than quicksort on average.
- Because of the above two facts, a relatively new hybrid sorting algorithm called "introspective sort" looks very promising. Introspective sort basically applies quicksort recursively until it realizes something has gone wrong then switches to heapsort. In this way it has a worst case performance of **O(n*ln(n))** and an average performance basically equal to quicksort.
- mergesort is guaranteed to run in **O(n*ln(n))** but requires n units of additional temporary storage.

Now when thinking about all of this, what struck me is that I could not *see* really why heapsort is slower than quicksort. And I've not heard or read a credible explanation for this either.

Update: David MacKay explains it here.

QuickSort

For large enough n, some subparitions of will contain small pathological cases. Furthermore, the partitioning breakdown will be arbitrary and thus necessarily not be even and therefore will perform more like **n*(lg(n)+eps)** comparisons/swaps. Work arounds like "median of 3" are not convincing, and just shifts the pathological cases. Balancing this out is the fact that quick sort is just a context insensitive recursive algorithm -- so its limited mostly by the speed at which it sorts the very smallest sub-lists (for each partition of size **m** processed there are two other partitions of smaller size that are processed.) By putting in a trivial check for small lists and performing them with explicit hand coded optimal sorts will boost quick sort's overall performance.

HeapSort

Heap sort is based around the idea of arranging the elements into a "heap" which allows an **O(1)** algorithm for finding the maximum element and an **O(ln(n))** for deleting it out of the array. By expressing the heap in clever way, this can all be mapped in-place with the original array. One common error made in implementing heap sort is by breaking the operations down all the way to paired element swaps. The operation of re-establishing the heap property of a heap after its maximum element is deleted (called sifting) is essentially a ***rotation*** of a sequence of sub-heap maximums. As I showed in my article about in-place array rotation, rotations require only one extra temporary and one read and write for each position of the cycle to be rotated. This leads to nearly a 2x improvement on the number of required reads/writes to the array. I was not able to find anyone else's reference implementation which took this into account.

But otherwise, heapsort doesn't have any substantial number of interesting degenerate cases (other than the infrequent short rotation) to leverage its overall performance.

MergeSort

Mergesort kind of combines the good features of quicksort and heapsort. It is also context insentively recursive, so small cases are a major leverage point for overall performance as well. However, it also is recursive in a static way, so that it cannot vary or be caught in bad cases where it might perform uncharacteristically badly. The problem with Mergesort is that every element will move **lg(n)** steps (or perhaps a few less because of the exploitability of direct sorting of small sub-lists) regardless. So the average, worst and best case running time is identical no matter what.

Measurements

Now given all this information its a bit hard to see which of these should really be fastest in real life. That leaves us with one final possibility -- actually try it out in practice. Below are the results from just such a test:

	Athlon XP 1.620Ghz				**Power4 1Ghz**
	Intel C/C++ /O2 /G6 /Qaxi /Qxi /Qip	**WATCOM C/C++** /otexan /6r	**GCC** -O3 -march=athlon-xp	**MSVC** /O2 /Ot /Og /G6	**CC** -O3
Heapsort	2.09	4.06	4.16	4.12	16.91
Quicksort	2.58	3.24	3.42	2.80	14.99
Mergesort	3.51	4.28	4.83	4.01	16.90

Data is time in seconds taken to sort 10000 lists of varying size of about 3000 integers each. Download test here

I suspected that heapsort should do better than its poor reputation and I think these results bear that out. Only MSVC really shows it to be more than 20% slower than quicksort and the always awesome Intel compiler turns in the best overall score with heapsort (by a *large* margin.)

It turns out that much of the reason for Intel's large margin of victory are the /Qaxi /Qxi flags used to build the test. The inner loop for heapsort has an opportunity to perform conditional computation rather than an unpredictable conditional branch (a big no no on modern microprocessors) and these flags allow the Intel compiler to exploit this situation while the other compilers were unable to do this. The quicksort algorithm inner loop cannot be predicated, and the Intel compiler did not do it with mergesort (even though it is possible to do it.)

The academics commonly cite comparison and swap counts. Of course, with my implementation of heapsort there are no *swaps* per se. So its more comparable to count the number of reads and writes. So lets count them up:

	Comparisons	**Reads/Writes**
Heapsort	61045.4	40878.2
Quicksort	22037.8	16322.5
Mergesort	31755.0	31225.0

Data is average number of operations taken to sort 1000 lists of varying size of about 3000 integers each.

Ok, this is appears to be the source of the conventional wisdom. Indeed, quicksort will perform almost 3 times better than heapsort and almost 2 times better than mergsort. (It should be noticed that my results show heapsort to be far better here than results obtained by others who count swaps which show relative differences closer to 10 times.) So whenever the performance is dominated by comparison or memory bandwidth costs quicksort really *is* the best choice.

This will be the case when using generic callback mechanisms like qsort() does, or when dealing with generic objects like STL does.

In the face of these counts, how is it possible for heapsort to do as well as it does in my previous measurements? Its not explained just by compiler efficiency. For this we have to really dive in under the hood of these algorithms and understand how this relates to modern microprocessor performance. The key considerations are parallelism and branching penalties.

Heapsort at a low level

Between each extraction of maximum elements in heap sort, a sift down operation occurrs that takes **lg(i)** steps where i varies from 1 to n. This sift down operation is basically a loop which traverses the binary tree structure of the heap always chosing the node with the higher value and shuffling these nodes upward. The direction down the tree that this operation goes each time is basically unpredictable, but at its lowest level manifests as adding either 0 or 1 to the current node index. For a modern compiler and underlying microprocessor this allows it to convert the comparison to this computation without a branch -- so the reduces the branch penalties to just one at the loop exit. For older compilers/microprocessors, there are **lg(i)** unpredictable branches. So in total there are about **lg(1)+lg(2)+...+lg(n)** (which is less than **n*lg(n)**) unpredicatable branches for older compilers and basically just **n** unpredictable branches for newer compilers/CPUs.

The operations in each instance of the sift down loop are all pipelinably parallel with respect to each other without a lot of intervening branch control. What this means is that for super-scalar, out of order, post modern microprocessors as instructions from one pass through the loop executing instructions from subsequent instances can actually start executing in parallel. So while copying the previous node upwards, the CPU can determine which of the next two nodes is the correct path to follow in parallel.

Quicksort at a low level

For each scanned element there is an unpredictable branch. So there are roughly **n*lg(n)** mispredicted branches. However, there is no simple way to translate these to conditional computations. This is mitigated somewhat by leveraging optimal sorting for the smallest sub-lists.

Each operation inside the paritioning step is essentially directly dependent on the results of the previous one. Because of this, there is almost no real parallelism available.

Mergesort at a low level

Mergesort actually requires somewhere between 1 and 3 predictable branches per read/write operation. It starts out mostly requiring 1, but will degenerate to closer to 3 as each half gets smaller. So the result is between **n*lg(n)** and **3*n*lg(n)** unpredictable branches. Although it is possible for a modern CPU to use conditional computation instructions in mergesort as well, the Intel C/C++ did not do it. Some experimentation by hand with it myself indicates that trying to force such a thing creates an additional indirection (aka "store to load forwarding") problem which makes the loop even slower. So conditional computation instructions don't help in this case.

As to parallelism, the conditional is dependent on the results from the store and variable increment, so there is no opportunity for mergesort either.

Conclusion and future work to be considered

Where modern processor and compiler considerations dominates, heapsort will actually *outperform* quicksort both theoretically and in practice. Only when the cost of comparisons and/or the operation of moving the entry itself starts to dominate will quicksort retake the lead.

This is something to consider when designing an abstract data type. If one has a list of pointers, or which can otherwise be satisfactorily sorted by indirect index, and it is essentially being sorted by a scalar (something desirable to for sort performance, regardless) then heapsort can be the most attractive option.

This leads us with one open question about a common sorting scenario. What if we want to sort text strings? Here the comparison performance is variable, and likely to have higher average cost than scalar comparison, but the read/write penalties will not necessarily change. While we expect quicksort to benefit from this, if the strings are totally random (which will lead to the roughly minimum cost for the string comparisons) would it be enough to tip the balance back in favor or quicksort when using the Intel compiler? This is unanswered.

Intel's compiler has quite an excellent reputation for delivering performance (established by many objective tests) and this has motivated competitors to try to improve their compilers. In this sense it is hoped that more compilers will deliver the same capabilities as Intel's does in the future. But as a consequence, heapsort will return as a serious candidate for raw sorting.

Update: Feedback

I have been contacted by various people who have challenged my results. Specifically, they claim that my quicksort implementation is unfairly sub-optimal. The complaints are roughly as follows:

1. *Instead of using the more typical "swap" operation, the inner loop of my quicksort contains gotos and appears more complicated, and thus must be slower.*

   This is pure nonsense. I started with a more typical "swap" loop and actually modified the loop in stages, measuring the results, and examining the compiler disassembly (from more than one compiler) every step of the way. I highly doubt that any prior effort has had comparable analysis.

   The result is my loop has roughly half as many reads/writes as the swap version. But the gotos also minimizes all control logic totally. This reduces the overhead for partitioning which changes other analysis.

2. *Once the partition size drops below a certain number (higher than 3) the algorithm should switch to Insert Sort (or some other O(n^2) sort).*

   I tried to do this, and in fact automatically *searched* for the optimal value for such a cutoff. The results were different for different compilers (which reflects the importance of the overhead of the pivot part of the loop) but no result showed a relevant improvement over the version shown.

   It should be noted that sort3 and sort2 are **optimal** (for our random data situation) sort algorithms. That means, the disadvantage for using more partitioning (versus an earlier cutoff to InsertSort style solution) is offset by the advantage of using these optimal sorts. sort3 and sort2 are many many times faster than InsertSort on 3 or 2 elements. But, InsertSort is an O(n^2) algorithm, so its performance on 6 elements is 4 times slower than its performance on 3 elements etc. So for an InsertSort cut over to be a better solution it needs to outperform the partition speed (since it cannot hold a candle to the sort3/sort2 functions.) For large n it has no chance of doing this, and considering the improved partition loop as described above, it will have a hard time beating the partition code as well. So this result should not come as a surprise at all.

3. *The number of elements of each array is too small.*

   Yes this is true -- I picked something that would be sure to fit into the on-chip CPU cache of any modern CPU. The reason for this is that the Comparisons and Reads/Writes table that is shown above already tells us what's going to happen. As we add cost to read/writes, Quicksort will clearly be a better solution. Furthermore, heapsort has very poor data locality, while Quicksort has excellent locality, which means the read/write penalties will be worse for heapsort than Quicksort. Under these circumstances its clear that heapsort will lose if the data doesn't all fit into the on-chip cache of the CPU.

But this leads us to an "Introsort" style of hybrid algorithm. Introsort is just a quicksort that calls itself recursively as normal until its stack depth exceeds a certain threshold and just switches to heapsort for such bad partitions. Well in addition, the quicksort could also switch to heap sort once the partition's size falls below that of the L1-cache (and if we are using the Intel compiler) which would make the algorithm's constant factor over O(n*lg(n)) really be limited by heapsort efficiency, not quicksort efficiency.

Besides missing the above things, others who have written to me have tried to compare this to sorting floating point (increasing the comparison cost), sticking with SWAP() as the lowest level data movement primitive (just plain dumb), and using slower languages (such as Basic) that are inappropriate for performance analysis.

Update: Feedback (2)

I have received some insightful feedback from James Barbetti, who has clearly studied this problem far beyond what you get from most academic study or from typical text books. He suggests 1) multiprocessor environments would behave differently (that should not surprise anyone), 2) there are ways of improving heapsort by running it the opposite direction (I have not confirmed this), 3) Quicksort can be guaranteed to use O(1+log(n)) of stack by not recursing the larger partition but rather by looping back to the top after adjusting the input parameters to match the larger partition.

I have updated the Quicksort loop with non-recursion on the larger partition, and added a 4 entry sort4() optimal sort. This improves the quicksort results marginally for most of the compilers, and significantly for the Intel compiler. I've updated the performance results in the table above and the test source code.