Heapsort, Quicksort, and Entropy

Numerous web–pages compare heapsort and quicksort.

Most of them say something along the lines of `both take an average time scaling as N log N, but A good implementation of QUICKSORT usually beats HEAPSORT in practice.'

Some take this folklore a bit further, giving quantitative details: `On average the number of comparisons done in HEAPSORT is about twice as much as in QUICKSORT, but HEAPSORT avoids the slight possibility of a catastrophic degradation of performance.'

But few seem to ask the question `why should heapsort use twice as many comparisons?' People spend a lot of effort on trying to `get the best of both worlds', making hybrid sorting algorithms such as `introspective sort', which applies quicksort recursively and occasionally switches to heapsort if the recursion depth gets big.

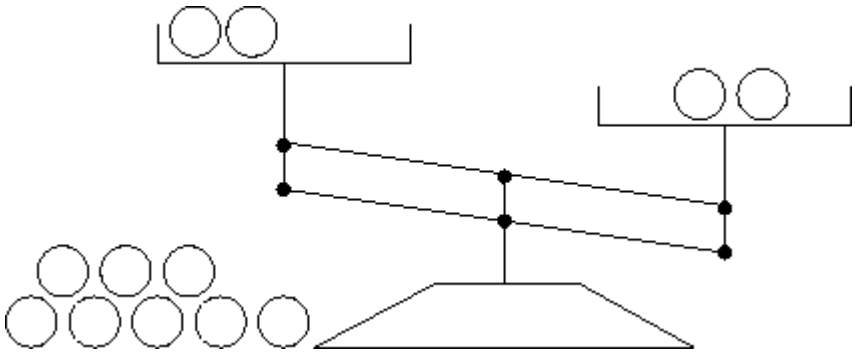
Quicksort and heapsort have been thoroughly compared by Paul Hsieh. He says `I suspected that heapsort should do better than its poor reputation and I think these results bear that out.' In his tests, the best compiler (for either heapsort or quicksort) produced a heapsort that was about 20% faster than quicksort, in total CPU time.

The total CPU tally is different from the number of comparisons made. Heapsort used an average of 61,000 comparisons, and Quicksort 22,000 comparisons, to sort lists of about 3000 objects. See his article for the explanation of the contrast between the comparison–count result and the CPU–time result.

The question I'd like to address, however, is, **why Heapsort uses more comparisons than quicksort**. Paul Hsieh says `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.'

I think there is a simple explanation, based on the idea of **expected information content**. To make this readable, let's ramble our way via a classic puzzle.

The weighing problem



You are given 12 balls, all equal in weight except for one that is either heavier or lighter. You are also given a two–pan balance to use. In each use of the balance you may put any number of the 12 balls on the left pan, and the same number on the right pan, and push a button to initiate the weighing; each weighing has three possible outcomes: either the weights are equal, or the balls on the left are heavier, or the balls on the left are lighter (the third case is shown in the figure above). Your task is to design a strategy to determine which is the odd ball *and* whether it is heavier or lighter than the others *in as few uses of the balance as possible*.

Many people find the solution to this puzzle by trial and error. (The odd ball can be identified in 3 weighings.) But if you find the final solution by trial and error, it feels rather complicated.

There is a better way.

To minimize the number of experiments made, we surely wish to maximize the average amount of *information gained* per experiment.

And Shannon showed that there is only one sensible way to define the expected amount of information to be gained from an outcome, namely the entropy of that outcome. (For definition of entropy and further discussion of the weighing problem, see my book: Information Theory, Inference, and Learning Algorithms.

We can solve the weighing problem in a jiffy by always selecting a measurement that has the maximum entropy; or to put it in probability terms, a measurement such that there are as many different conceivable outcomes as possible, and they all have as nearly as possible **equal probabilities**.

Sorting and bits

If the sort is being conducted using the information from binary comparisons alone, then the maximum average information that can be generated per comparison is 1 bit.

The amount of information required to sort N objects is exactly $\log_2 N!$ bits (assuming no prior information about the objects).

Using Stirling's approximation, this total information content is $T = N \log_2 N - N \log_2 e$.

The average number of comparisons required by any sorting algorithm can certainly not be less than T. And it will only approach T *if every comparison has a 50:50 chance of going either way*.

So, why is it that

`Heapsort is not quite as fast as quicksort in general'?

Surely it is because *heapsort makes comparisons whose outcomes do not have equal prior probability*. We'll see why this is so in a moment.

Incidentally, standard randomized quicksort has the same defect. It's irritating how all these algorithm–studiers just say `randomized quicksort uses O(N log N) comparisons on any input with very high probability', thus throwing away the fascinating constant factor in the average cost. Absurd `O' notation! What a silly pretense, to say that `we care about asymptotic performance for large N', as if that means that we don't care about the difference between a 4 N log N algorithm and a 1 N log N one!

It is so much more interesting if we go the extra mile and work out the factor multiplying N log N; or to put it another way, the base of the logarithm. Let me reveal the final result, then prove it. Randomized quicksort has an average cost of $N \log_{e^{1/2}} N$. That's the log to the base of the square root of e, which is the log to base 1.649.

This expected cost is greater than the ideal cost, T \simeq N $\log_2$N, by a factor of $1/\log_2 1.649 \simeq 1.39$. If we really care about comparison count, we should be looking for a

better algorithm than quicksort!

More here... You can see that quicksort has unbalanced probabilities by imagining the last few comparisons of numbers with a pivot. If the preceding 100 comparisons have divided 70 on one side and 30 on the other side, then it seems a good prediction that the next comparison with the pivot has a probability of 0.7 of going the first way and 0.3 of going the other.

Back to heapsort

Heapsort is inefficient in its comparison count because it pulls items from the *bottom* of the heap and puts them on the top, allowing them to trickle down, exchanging places with bigger items. This always struck me as odd, putting a quite–likely–to–be–small individual up above a quite–likely–to–be–large individual, and seeing what happens. Why does heapsort do this? Could no–one think of an elegant way to promote one of the two sub–heap leaders to the top of the heap?

How about this:

Modified Heapsort (surely someone already thought of this)

1. Put items into a valid max–heap
2. Remove the top of the heap, creating a vacancy 'V'
3. Compare the two sub–heap leaders directly below V, and promote the biggest one into the vacancy. Recursively repeat step 3, redefining V to be the new vacancy, until we reach the bottom of the heap.

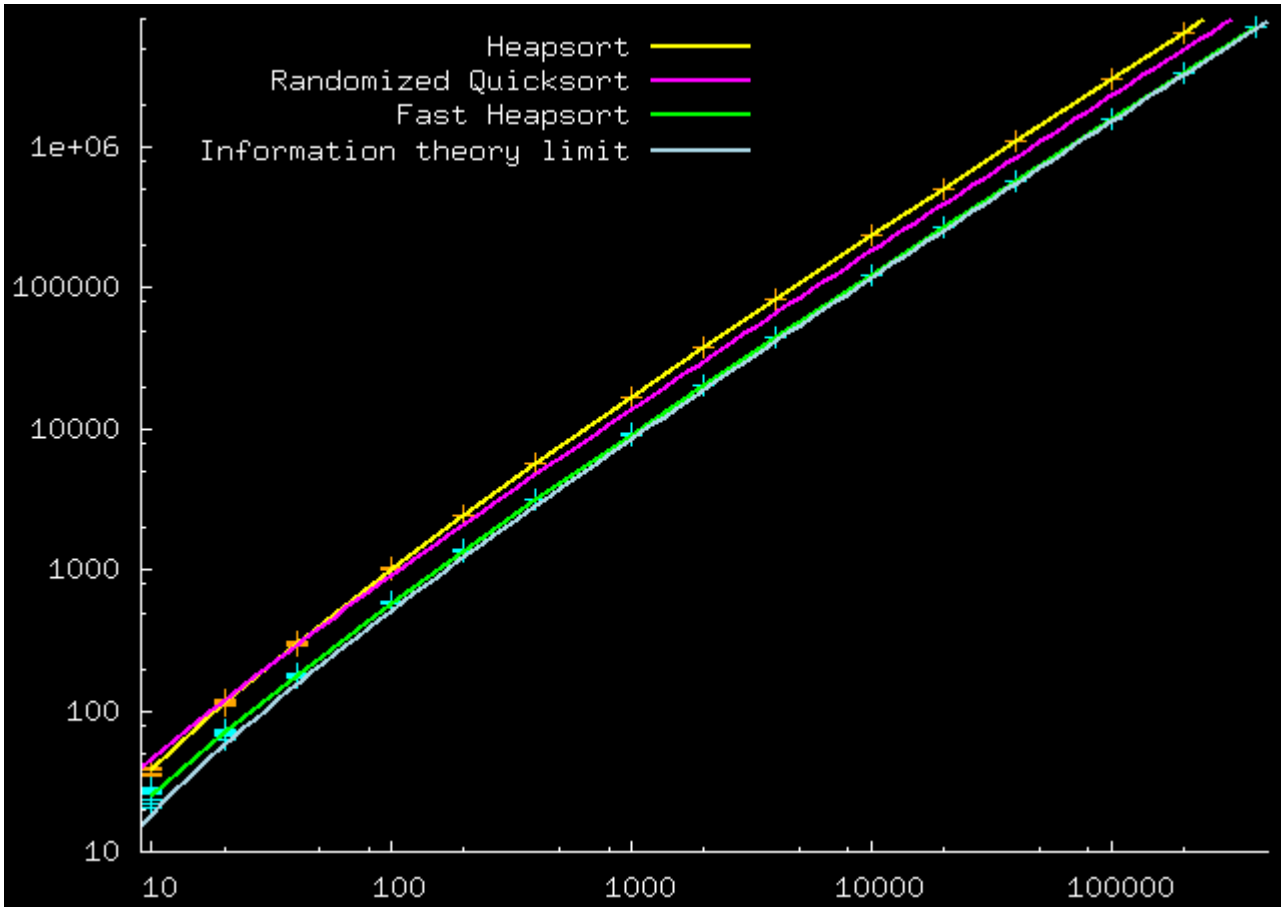
   (This is just like the **sift** operation of heapsort, except that we've effectively promoted an element, known to be the smaller than all others, to the top of the heap; this smallest element can automatically trickle down without needing to be compared with anything.)
4. Go to step 2

Disadvantage of this approach: it doesn't have the pretty in–place property of heapsort. But we could obtain this property again by introducing an extra swap at the end, swapping the 'smallest' element with another element at the bottom of the heap, the one which would have been removed in heapsort, and running another sift recursion from that element upwards.

Let's call this algorithm **Fast Heapsort**. It is not an in–place algorithm, but, just like Heapsort, it extracts the sorted items one at a time from the top of the heap.

Performance of Fast Heapsort

I evaluated the performance of Fast Heapsort on random permutations. Performance was measured solely on the basis of the number of binary comparisons required. [Fast Heapsort does require extra bookkeeping, so the CPU comparison will come out differently.]



Performance of Fast Heapsort.
Horizontal axis: Number of items to be sorted, *N*.
Vertical axis: Number of binary comparisons. The theoretical curves show the asymptotic results for Randomized Quicksort (2 N ln N) and the information–theoretic limit, log_2 N! simeq (N log N – N)/log 2.

I haven't proved that Fast Heapsort comes close to maximizing the entropy at each step, but it seems reasonable to imagine that it might indeed do so asymptotically. After all, Heapsort's starting heap is rather like an organization in which the top dog has been made president, and the top dogs in divisions A and B have been made vice–president; a similar organization persists all the way down to the lowest level. The president originated in one of those divisions, and got his job by sequentially deposing his bosses.

Now if the boss leaves and needs to be replaced by the best person in the organization, we'll clearly need to compare the two vice–presidents; the question is, do we expect this to be a close contest? We have little cause to bet on either vice–president, a priori. There are just two asymmetries in the situation: first, the retiring president probably originated in one of the two divisions; and second, the total numbers in those two divisions may be unequal. VP `A' might be the best of *slightly more* people than VP `B'; the best of a big village is more likely to beat the best of a small village. And the standard way of making a heap can make rather lop–sided binary trees. In an organization with 23 people, for example, division A will contain (8+4+2+1)=15 people, and division B just (4+2+1)=7.

To make an even–faster Heapsort, I propose two improvements:

1. Make the heap better–balanced. Destroy the elegant heap rules, that i's children are at (2i) and (2i+1), and that a heap is filled from left to right. Will this make any difference?

Perhaps not; perhaps it's like a Huffman algorithm with some free choices.

2. Apply information theory ideas to the initial heap–formation process. Does the opening Heapify routine make comparisons that have entropy significantly less than one bit?

Back to quicksort

We can give quicksort the entropy treatment too. Quicksort is wasteful because it persists in making experiments where the two possible outcomes are not equally likely. Roughly half of the time, quicksort is using a 'bad' pivot – 'bad' in the sense that the pivot is outside the interquartile range – and, once it has become clear that it's a bad pivot, every comparison made with that pivot yields an expected information content significantly less than 1 bit.

A simple hack that reduces quicksort's wastefulness is the 'median of three' modification of quicksort: rather than picking one pivot at random, three candidates are picked, and their median is chosen as the pivot. The probability of having a bad pivot is reduced by this hack. But we can do better than this. Let's go back to the beginning and analyse the information produced by quicksort.

When we pick a pivot element at random and compare another randomly selected element with it, the entropy of the outcome is clearly one bit. A perfect start. When we compare another element with the pivot, however, we instantly are making a poor comparison. If the first element came out `higher', the second is more likely to be higher too. Indeed the probability that the second is higher is roughly 2/3. (A nice Bayes' theorem illustration, I must remember that one! For N=3, and for N=4 objects, 2/3 is exactly right; perhaps it is exact for all N.) The entropy of (1/3,2/3) is 0.918 bits, so the inefficiency of this first comparison is not awful – just 8% worse than perfect. On entropy grounds, we should go and compare two other elements with each other at our second comparison. But let's continue using Quicksort. If the first *two* elements both turn out higher than the pivot, the next comparison has a probability of 3/4 of coming out `higher' too. (Entropy: 0.811 bits.) This situation will arise more than half the time.

Table 1 shows, after **5** elements have been compared with the pivot element, what the possible states are, and what the entropy of the next comparison in quicksort would be, if we went ahead and did it. There is a one–third probability that the state is either (0,5) or (5,0) (meaning all comparisons with the pivot have gone the same way); in these states, the entropy of the next comparison is 0.59 – which is 40% worse than the ideal entropy.

State		Probability of this state	Probability that next element will go left	Entropy
left	right			
0	5	1/6	1/7	0.59167
1	4	1/6	2/7	0.86312
2	3	1/6	3/7	0.98523
3	2	1/6	4/7	0.98523
4	1	1/6	5/7	0.86312

5	0	1/6	6/7	0.59167

Table 1

Table 2 shows the *mean* entropy of the outcome at each iteration of quicksort.

Iteration of quicksort	Expected entropy at this step	Minimum entropy at this step
0	1	1
1	0.918296	0.918296
2	0.874185	0.811278
3	0.846439	0.721928
4	0.827327	0.650022
5	0.81334	0.591673
6	0.80265	0.543564
7	0.794209	0.503258
8	0.78737	0.468996
9	0.781715	0.439497
10	0.77696	0.413817

Table 2

We can use calculations like these to make rational decisions when running quicksort. For example, we could give ourselves the choice between continuing using the current pivot, *or* selecting a new pivot from among the elements that have been compared with the current pivot (in a somewhat costly manner) and continuing with the new pivot. The 'median of three' starting procedure can be described like this: we always start by comparing **two** elements with the pivot, as in standard quicksort. If we reach the state (1,1), which has probability of 1/3 of happening, then we continue using the current pivot; if we reach the state (0,2) or (2,0), we decide this is a bad pivot and discard it. We select a new pivot from among the other two by comparing them and choosing the one that is the median of all 3. This switch of pivot has an extra **cost** of one comparison, and is expected to yield beneficial returns in the form of more bits per comparison (to put it one way) or a more balanced tree (to put it another way).

We can put the 'median of 3' method on an objective footing using information theory, and we can generalize it too.

Imagine that we have compared (M–1) of N items with a randomly selected pivot, and have reached the state (m1,m2) (i.e., m1 to the left and m2 to the right). We now have a choice to continue using the same pivot, which gives us an expected return of H_2(p) at the next comparison, where p = (m1+1)/(m1+m2+2), or we could invest in a median–finding algorithm, finding the median of the M items handled thus far. (Median can be found with an expected cost proportional to M; for example, quickselect's cost is about 4M.) We can evaluate the expected return–to–cost ratio of these two alternatives. If we decide to make (N–M) more comparisons with the pivot the expected return is roughly R = (N–M)H_2(p) bits. [Not exactly right; need to do the integrals to find the exact answer.] If we switch to a new pivot then

proceed with quicksort, discarding the information generated while finding the new pivot at subsequent iterations (which is wasteful of course – and points us to a new algorithm, in which we first sort a subset of elements in order to select *multiple* pivots), the cost is roughly (N–M)+4(M–1) comparisons and the expected return is roughly R' = (N–M)H_2(p') bits, where p' is the new pivot's rank.

If we approximate R' \simeq N–M then finding the new pivot has the better return–to–cost ratio if

$$(N–M)/ ((N–M)+4(M–1)) > H_2(p)$$
$$\text{i.e. } 1/ (1+4(M–1)/(N–M)) > H_2(p)$$

As we would expect, if the number of future points to be compared with the pivot, N–M, is large, then we feel more urge to criticise the current pivot.

Further modifying quicksort, we can plan ahead: it's almost certainly a good idea to perfectly sort M randomly selected points and find their median, where M is roughly sqrt(N/log(N)), and use that fairly accurate median as the pivot for the first iteration.

Summary: 'median–of–3' is a good idea, but even better (for all N greater than 70 or so) is 'median–of–sqrt(N/log(N))'. If you retain the sorted subset from iteration to iteration, you'll end up with something rather like a Self–balancing binary search tree.

References

Nice explanation of randomized median–finding in O(n) time and deterministic median–finding.

David MacKay December 2005

This file was originally at http://www.aims.ac.za/~mackay/sorting/sorting.html
and is now at http://www.inference.org.uk/mackay/sorting/sorting.html