Sorting is removing 'inversions'.

In an array sorted by ($\leq$) we have $\forall i, j\left(0 \leq i < j < n \rightarrow A[i] \leq A[j]\right)$. If the array isn't completely sorted, it will contain ***inversions***. That is, $\exists i, j\left(0 \leq i < j < n \wedge A[i] > A[j]\right)$.

Sorting can be seen as a process of removing those inversions, by exchanging (permuting) elements.

Suppose that `x<->y` is an 'exchange' instruction, which swaps the contents of two variables or array elements.

Here's insertion sort again:

```
for (int i=1; i<n; i++) {
  for (int j=i; j!=0 && A[j-1]>A[j]; j--)
    A[j-1]<->A[j];
}
```

Each step of the inner loop removes exactly one inversion.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

This program always does n comparisons `A[j-1]>A[j]`, plus, for each inversion it finds, an exchange `A[j-1]<->A[j]` and another comparison.

In general, for an array of size N containing I inversions, it takes time proportional to $N + I$.

And thus insertion sort is $O(N)$ in a sorted array (because $I = 0$), and 'runs quickly' (Weiss) on an 'almost sorted' array (one in which I is small).

Weiss (p226) shows that the number of inversions in an array is at most $N(N-1)/2$, and on average is $N(N-1)/4$. This explains why insertion sort is $O(N^2)$ both in the worst case and on average.

> *Weiss's argument doesn't work if the array contains duplicate elements. Why not?*

Shellsort: a better insertion sort.

Named for its inventor, Donald Shell. Really.

Insertion sort does a fixed amount of work to remove each inversion.

Shell sort can remove several inversions with one exchange.

Its running time is proved to be $O\left(N^{3/2}\right)$ in the worst case, and seems on average to be $O\left(N^{5/4}\right)$ (Weiss p230). It might even be as good as $O\left(N^{7/6}\right)$

But nobody is quite sure. Really.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Consider the 'h-chains' ($h \geq 1$) of $A[0..n-1]$:

$A[0], A[h], A[2h], \ldots$

$A[1], A[h+1], A[2h+1], \ldots$

$A[2], A[h+2], A[2h+2], \ldots$

...

$A[h-1], A[2h-1], A[3h-1], \ldots$

The chains are disjoint.

> ***disjoint**: share no elements. Technical language.*

We can sort them, if we wish, separately.

Why should we wish? Well:

> it is a lot quicker to sort h little h-chains than one large array ($h \times (N/h)^2 = N^2/h$);
>
> each re-arrangement in a chain has a chance of removing a lot of inversions between the elements moved and elements in other chains.

Together these give us the magic.

We sort the h-chains with large spacing h, then we reduce h a bit and do it again, then reduce again and sort again, ... , then finally sort with $h = 1$ on the whole array.

Here's a partial description of the algorithm. `STARTINGGAP` and `NEXTGAP` are parameters, which I will discuss later:

```
for (int h = STARTINGGAP; h>0; h=NEXTGAP) {
  for (int i = h; i<n; i++) {
    for (int j=i; j>=h && A[j]>A[j-h]; j-=h)
      A[j-h]<->A[j];
  }
}
```

The elements in $A[0..h-1]$ are each the first in their chains, so we start with $i = h$. Then each value of i in $h, h+1, h+2, ..., n-1$ is a position in an h-chain, and we insertion-sort an element into that chain (lines 3 and 4).

We get different efficiency, depending on we write in place of `STARTINGGAP` and `NEXTGAP`. It's subtle, and we have to rely on experts for information.

> *If you see yourself as a computer science professor one day, this might be a point to start! Try following up the references in Weiss, and try reading some of the proofs for yourself.*

Weiss (p230) points out that if we write

```
for (int h=n/2; h>0; h=h/2)
```

then there is a possible worst case which gives $O\left(N^2\right)$ performance.

He then states that if you write a loop which starts with $n \div 2$ and divides by 2 at each step, but adds 1 if you get an even answer, you get $O\left(N^{3/2}\right)$ in the worst case and $O\left(N^{5/4}\right)$ in the average case.

```
for (int h=n/2; h>0; h=h/2, h=h-h%2+1)
```

> *At present there is no proof of the average-case result. It's an experimental result.*

If you divide by 2.2 instead of 2, you get even better performance – $O\left(N^{7/6}\right)$ – and nobody knows why! The algorithm is on p229.

How can it work so well?

An array in which the *h*-chains are sorted, for some value of *h*, is ***h*-sorted**. So an array might be 5-sorted, or 3-sorted, or 17-sorted, or all the above.

Fact about Shellsort: when you take an *h*-sorted array and *j*-sort it ($j < h$), you finish up with something that is both *j*-sorted and *h*-sorted.

> *I think I can see why this is, but I'm not going to put it on a slide yet.*

The effect is to produce a final sort which is just insertion sort (because $h = 1$), which is fast because most of the inversions have already gone.

> *Note that the formulæ* `STARTINGGAP` *and* `NEXTGAP` *have to be designed to produce a sequence* $h_k, h_{k-1}, \ldots, h_0$ *which ends with* $h_0 = 1$, *otherwise the sort won't complete properly.*

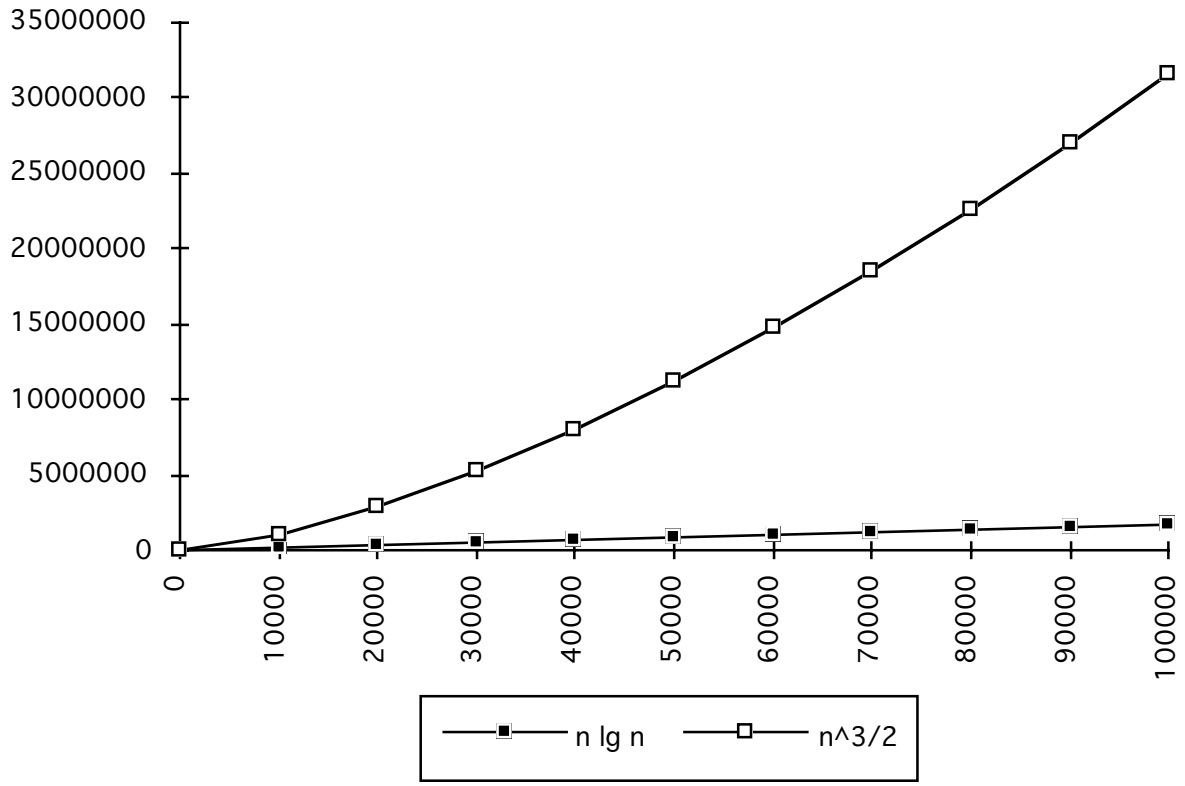
Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

The performance of Shellsort.

In the next batch of slides there is a health warning about the kind of graphs shown on the next few slides. You have to consider 'constants of proportionality' before you can compare different algorithms, not just the magnitude of the formulae in an $O(...)$ judgement.

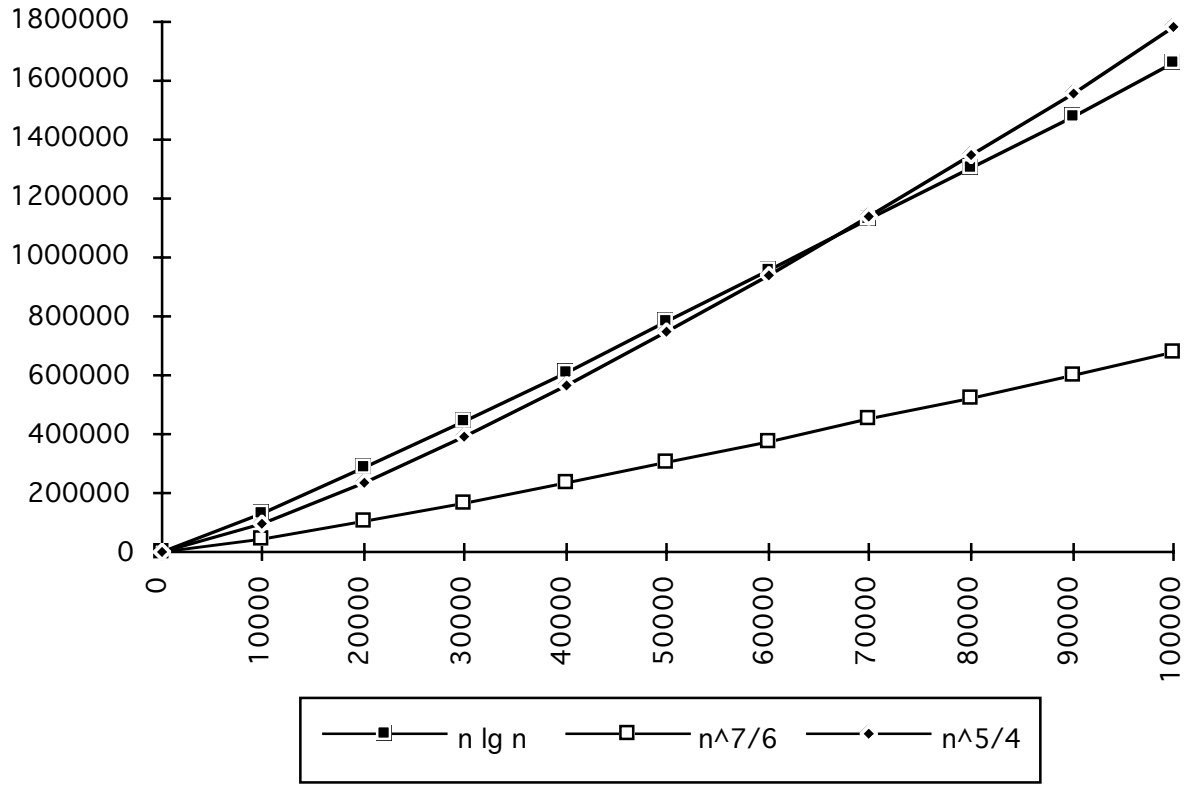
But it's still interesting to see how the different formulæ behave.

$N^{3/2}$ (Shellsort worst case) grows much more quickly than $N \lg N$ (we shall soon see some fancy sorting algorithms which run in $N \lg N$ time):

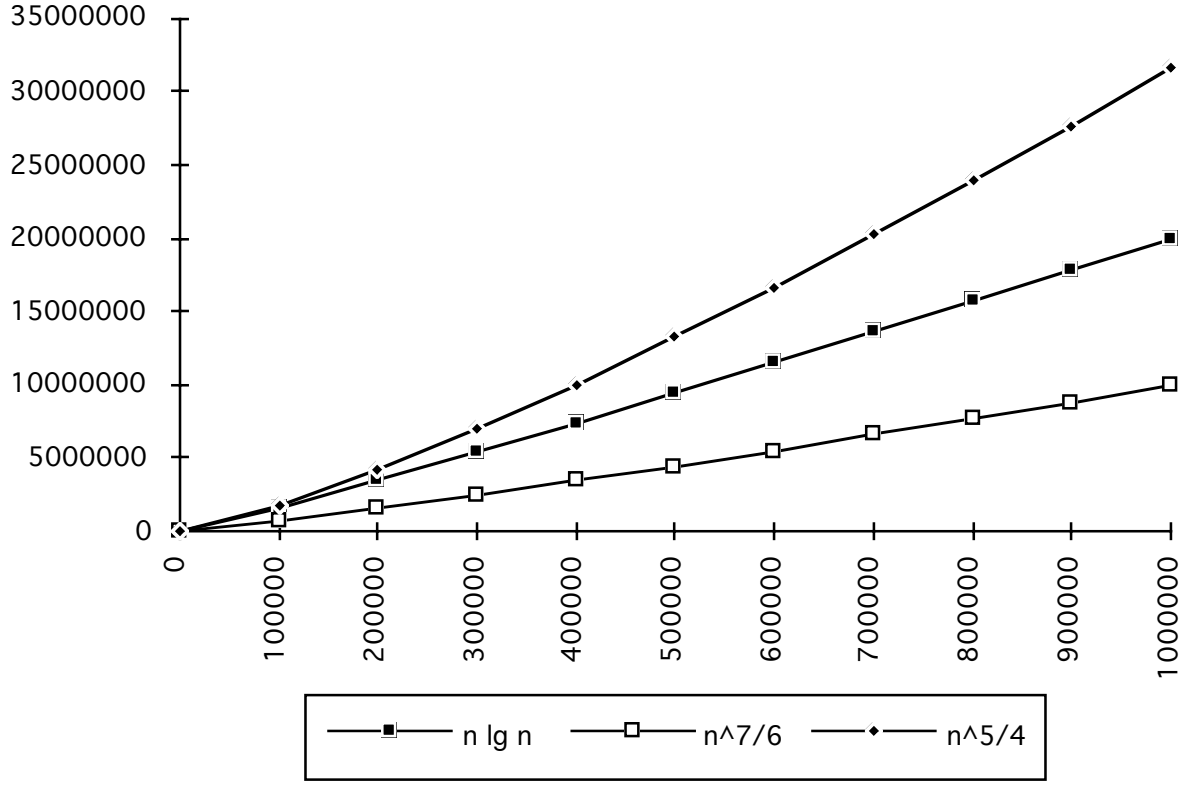


Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

But you have to go to quite large numbers before $N \lg N$ is smaller than $N^{5/4}$ (Shellsort average time, using halving to odd numbers):

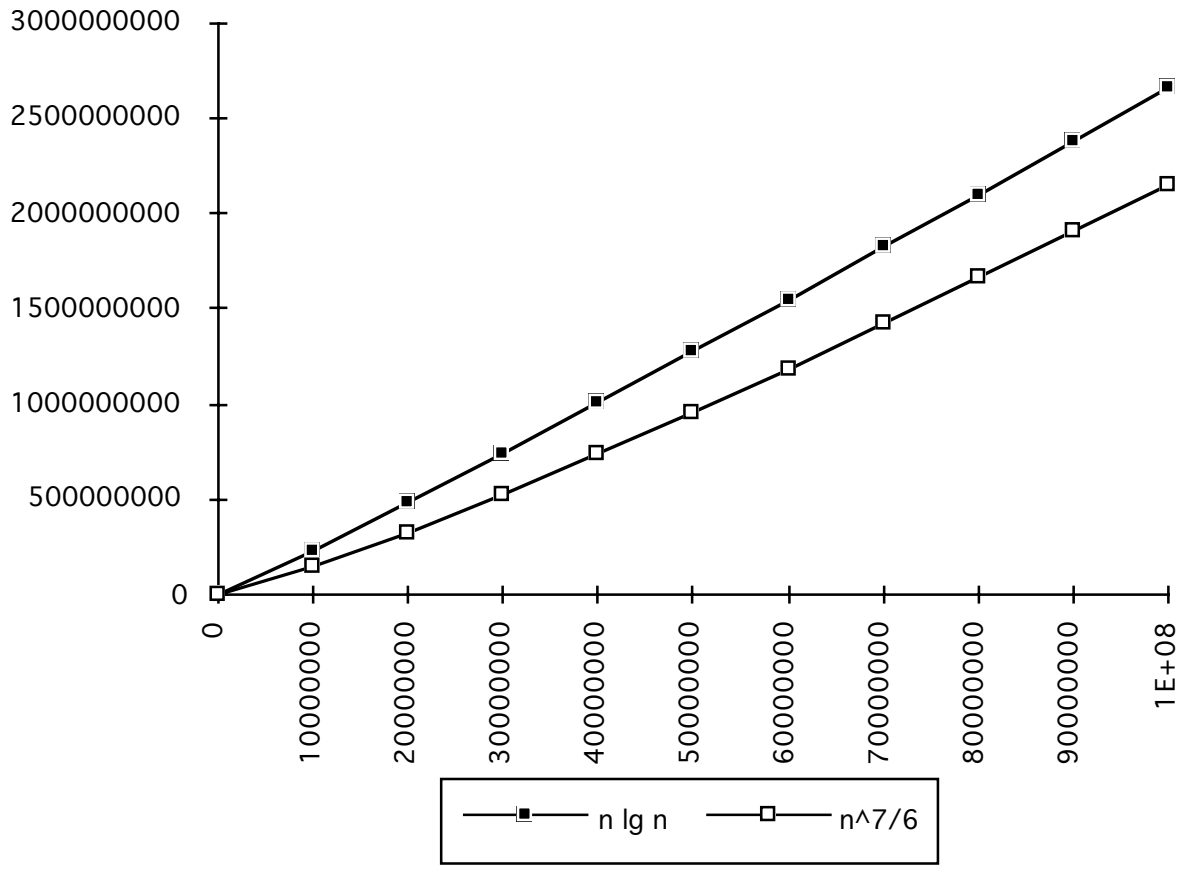


After that point $N^{5/4}$ grows more quickly than $N \lg N$, so the latter's advantage grows.

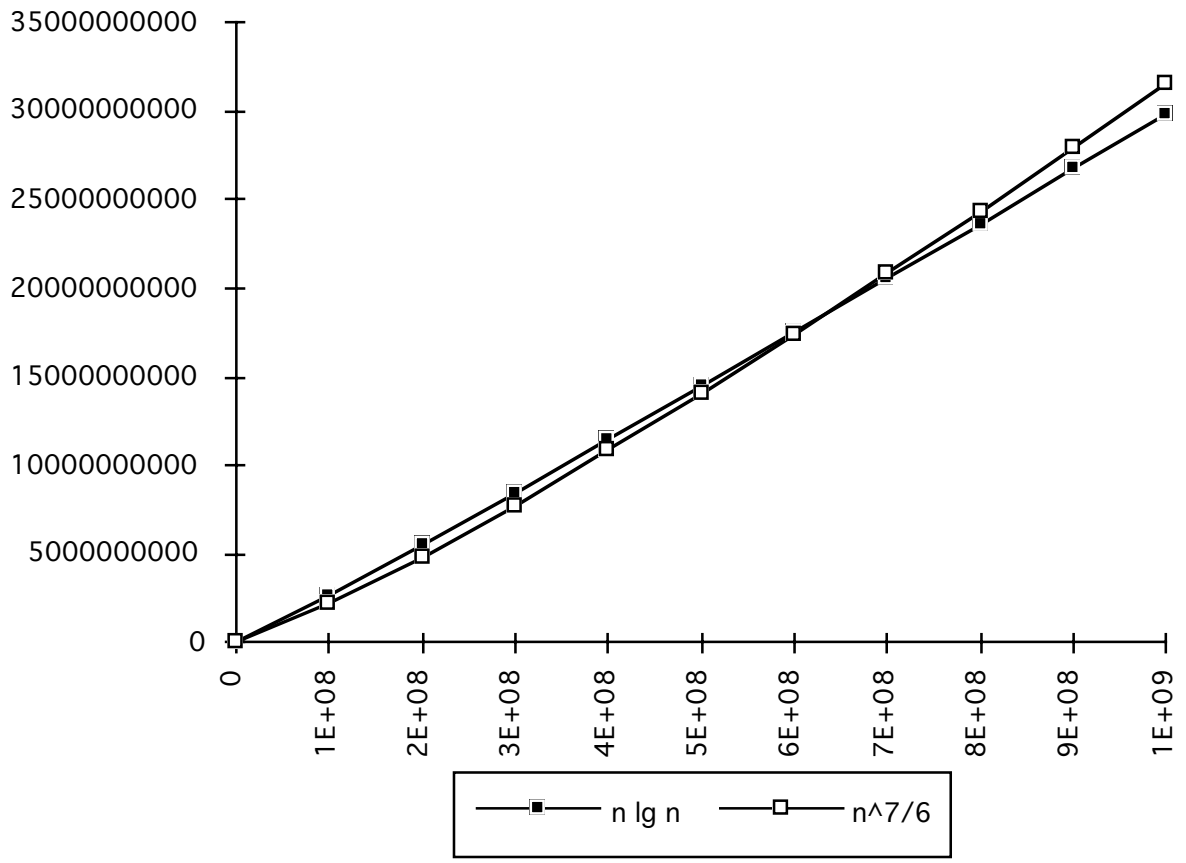


But look at $N^{7/6}$!

Even at a value of one hundred million, $N^{7/6}$ (Shellsort average time, using divide by 2.2) hasn't overtaken $N \lg N$ yet:



It happens eventually, as it must, somewhere round about half a billion:



What these graphs tell us is that Shellsort may be a serious rival to the $N \lg N$ algorithms which we shall consider later.

But experiment rules everything. To see which is faster, we shall have to conduct program races.

Actually, ***you*** shall conduct the races, in the lab.