In the worst case, Shell's increments give quadratic behavior.

When Shell's increments are used, the worst case can be proven to be $O(N^2)$. This bound is achievable if N is an exact power of 2, all the large elements are in even-indexed array positions, and all the small elements are in odd-indexed array positions. When the final pass is reached, all the large elements will still be in the even-indexed array positions, and all the small elements will still be in the odd-indexed array positions. A calculation of the number of remaining inversions shows that the final pass will require quadratic time. The fact that this is the worst that can happen follows from the fact that an h_k-sort consists of h_k insertion sorts of roughly N/h_k elements. Consequently, the cost of each pass is $O(h_k(N/h_k)^2)$, or $O(N^2/h_k)$. When we sum this over all the passes, we obtain $O(N^2\sum 1/h_k)$. Since the increments are roughly a geometric series, the sum is bounded by a constant. The result is a quadratic worst-case running time. One can also prove via a complex argument that when N is an exact power of 2, the average running time is $O(N^{3/2})$. Thus, on average, Shell's increments give a significant improvement over insertion sort.

If consecutive increments are relatively prime, the performance of Shellsort is improved.

A minor change to the increment sequence can prevent the quadratic worst case from occurring. If we divide `gap` by 2 and it becomes even, then we can add 1 to make it odd. We can then prove that the worst case is not quadratic but only $O(N^{3/2})$. Although the proof is complicated, the basis for it is that in this new increment sequence, consecutive increments share no common factors (whereas in Shell's increment sequence they do). Any sequence that satisfies this property (and whose increments decrease roughly geometrically) will have a worst-case running time of at most $O(N^{3/2})$.[1] The average performance of the algorithm with these new increments is unknown but seems to be $O(N^{5/4})$, based on simulations.

Dividing by 2.2 gives excellent performance in practice.

A third sequence, which performs well in practice but has no theoretical basis, is to divide by 2.2 instead of 2. This appears to bring the average running time to below $O(N^{5/4})$ (perhaps to $O(N^{7/6})$), but this is completely unresolved. For 100,000 to 1,000,000 items, it typically improves performance by about 25 to 35 percent, although nobody knows why. A Shellsort implementation with this increment sequence is coded in Figure 8.4. The complicated code at line 8 is necessary to avoid prematurely setting `gap` equal to 0. If that happens, then the algorithm is broken because we never see a 1-sort. Line 8 ensures that if `gap` is 2 and is about to be set to 0, it is reset to 1.

The table in Figure 8.5 compares the performance of insertion sort and Shellsort, with various gap sequences. These results were obtained on a reasonably fast machine. The test is clearly biased against the original gap sequence because N is chosen to be 125 times an exact power of 2. Thus, rounding up to an odd number is particularly beneficial, especially as N gets large. We could easily conclude that Shellsort, even with the simplest gap sequence, provides a significant improvement over the insertion sort, at a cost of little additional code complexity. A simple change to the gap sequence can further improve performance.

1. To appreciate the subtlety involved, note that subtracting 1 instead of adding 1 does not work. For instance, if N is 186, the resulting sequence is 93, 45, 21, 9, 3, 1, all of which share the common factor 3.