Shell sort

Invented by Donald Shell, 1959
http://krypton.mnsu.edu/~geneal/shell-donald_bio.htm

- Improves on insertion sort.
- Starts by comparing elements far apart, then elements less far apart, and finally comparing adjacent elements (effectively an insertion sort). By this stage the elements are sufficiently sorted that the running time of the final stage is much closer to O(N) than O(N^2).

Shell sort, is also known as the diminishing increment sort.
The idea of Shell sort is the following:

a) arrange the data sequence in a two-dimensional array
b) sort the columns of the array

The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column.

In each step, the sortedness of the sequence is increased, until in the last step it is completely sorted. However, the number of sorting operations necessary in each step is limited, due to the presortedness of the sequence obtained in the preceding steps.

Example:

Let 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2 be the data sequence to be sorted. First, it is arranged in an array with 7 columns (left), then the columns are sorted (right):

\[
\begin{align*}
3 & 7 & 9 & 0 & 5 & 1 & 6 \\
8 & 4 & 2 & 0 & 6 & 1 & 5 \\
7 & 3 & 4 & 9 & 8 & 2 & \quad \quad \rightarrow & 3 & 3 & 2 & 0 & 5 & 1 & 5 \\
& 7 & 4 & 4 & 0 & 6 & 1 & 6 \\
& 8 & 7 & 9 & 9 & 8 & 2 &
\end{align*}
\]

Data elements 8 and 9 have now already come to the end of the sequence, but a small element (2) is also still there. In the next step, the sequence is arranged in 3 columns, which are again sorted:
Now the sequence is almost completely sorted. When arranging it in one column in the last step, it is only a 6, an 8 and a 9 that have to be moved a little bit to their correct positions.

**Implementation** Actually, the data sequence is not arranged in a two-dimensional array, but held in a one-dimensional array that is indexed appropriately.

The algorithm uses an increment sequence to determine how far apart elements to be sorted are:

\[ h_1, h_2, \ldots, h_t \text{ with } h_1 = 1 \]

At first elements at distance \( h_t \) are sorted, then elements at distance \( h_{t-1} \), etc, until finally the array is sorted using insertion sort (distance \( h_1 = 1 \)).

An array is said to be \( h_k \)-sorted if all elements spaced a distance \( h_k \) apart are sorted relative to each other.

Shellsort only works because an array that is \( h_k \)-sorted remains \( h_k \)-sorted when \( h_{k-1} \)-sorted. This means that subsequent sorts with a smaller increment do not undo the work done by previous phases.

The **correctness of the algorithm** follows from the fact that in the last step (with \( h = 1 \)) an ordinary Insertion Sort is performed on the whole array. But since data are presorted by the preceding steps (\( h = 3, 7, 21, \ldots \)) only few Insertion Sort steps are required.
int j, p, gap;
comparable tmp;

for (gap = N/2; gap > 0; gap = gap/2)
    for (p = gap; p < N ; p++)
        {
            tmp = a[p];
            for (j = p; j >= gap && tmp < a[j-gap]; j = j - gap)
                a[j] = a[j-gap];
            a[j] = tmp;
        }

Animations:
http://www.sorting-algorithms.com/shell-sort
http://www.cs.pitt.edu/~kirk/cs1501/animations/Sort2.html

What should the increment sequence be?
There are many choices for the increment sequence. Any sequence that begins at 1 and always increases will do, although some yield better performance than others:
1. Shell's original sequence: N/2 , N/4 , ..., 1 (repeatedly divide by 2);
2. Hibbard's increments: 1, 3, 7, ..., 2^k - 1 ; k = 1, 2, ...
3. Knuth's increments: 1, 4, 13, ..., (3^k - 1) / 2 ; k = 1, 2, ...
4. Sedgwick's increments: 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ....

The sequence is obtained by interleaving the elements of two sequences:
1, 19, 109, 505, 2161,..... 9 (4^k – 2^k) + 1,   k = 0, 1, 2, 3,...
5, 41, 209, 929, 3905,.....2^{k+2} (2^{k+2} – 3 ) + 1, k = 0, 1, 2, 3, ...

Analysis
A Shellsort's worst-case performance using Hibbard's increments is Θ(n^{3/2}).
The average performance is thought to be about O(n^{5/4}).

The exact complexity of this algorithm is still being debated.

Experience shows that for mid-sized data (tens of thousands elements) the algorithm performs nearly as well if not better than the faster (n log n) sorts.