

Shell Sort

Shell sort is named after Donald Shell, who first published an account of this sorting technique. Shell sort is also called the *diminishing increment* sort, and is a variation on the basic insertion sort algorithm. In insertion sort, comparison is always done between adjacent elements (a_i, a_{i+1}), $i < n$, and a swap occurs if $a_{i+1} < a_i$. So, at most 1 inversion is eliminated for each comparison done with insertion sort.

The variation used in shell sort, is to avoid comparing adjacent elements until the last step of the algorithm. So, the last step of shell sort is effectively the insertion sort algorithm, but by the time we reach the last step, the list is either already sorted, or mostly sorted, so insertion sort runs in almost linear time (we already saw that the insertion sort algorithm is $O(n)$ if a list of n elements is already sorted in ascending order).

Shell sort runs in subquadratic time and works well in practice for lists of several thousand entries (upto 250,000). The algorithm is also very simple to implement, so shell sort is commonly used in practice for sorting basic lists of keys (e.g., integers).

The running time of shell sort is highly dependent on the selection of an increment sequence h_t, h_{t-1}, \dots, h_1 that effectively partition the original list into a set of sublists, which are sorted after one pass through the entire list. The next increment is used to partition the list into a new set of sublists, which are again sorted by a single pass through the entire list, and so on until increment h_1 , which is equal to 1. So, the final increment means we have just one large list, and so for increment h_1 , an insertion sort is done on the list.

Shell sort works better than insertion sort in practice because it “knocks out” at least 1 inversion per comparison, and very often several inversions. When all the inversions are eliminated, the list is sorted, so the real job of a sorting algorithm is to eliminate inversions.

Shell Sort Example

Given a list of 10 elements, the starting increment is 4, which partitions the list into 4 sublists, each with approximately 2 entries:

3 9 7 2 8 10 6 5 1 4

Q: What are the inversions in the list?

Shell Sort Increments

An exact analysis of the average running time of shell sort has not been completely achieved. However, in practice the algorithm has been shown to run in $O(n^{3/2})$ or $O(n(\lg n)^2)$, depending on how the increment sequence is defined. A poor choice for the increment sequence will yield $O(n^2)$ behavior in the worst case.

The definition of the increment sequence is critical to achieving subquadratic running time. The increment sequence that seems to work well in practice is: $\dots, 1093, 364, 121, 40, 13, 4, 1$, but no one has yet given a strong mathematical argument for why this sequence works better than other sequences. The sequence is defined by $h_i = (3^i - 1)/2$, for $1 \leq i \leq t$.

This starting element of the sequence, h_t , is easy to compute using a simple loop:

```
int h = 1;
while (h <= n)
    h = 3 * h + 1;
```

At the end of this loop, h has the highest increment in the sequence, h_t , which is less than or equal to the size of the list n . We use this increment to make the first partition of the list into h sublists, each with approximately n/h entries, and sort each sublist. We then need to compute h_{t-1} , generate a new partition of the list, and sort the resulting sublists, and so on until h_1 , which means $h == 1$.

It is not necessary to store the entire increment sequence at the start of the program, as the increments can be computed in reverse, using:

```
h = (h-1) / 2;
```

Shell Sort Example

Shell Sort Example

8/5/00

5 of 8

The Shell Sort Algorithm in C++

We use a template function that takes an array of any type T, containing n elements.

```
template<class T>
void shellSort (T a[], int n)
{
    int h = 1;

    while ( h <= n)
        h = 3 * h + 1; // compute first increment past the starting increment

    do {
        h = (h - 1) / 3; // compute the diminishing increment

        for (int i = h; i < n; i++) {
            T v = a[i];

            for (int j = i; a[j-h] > v && j >= h; j -= h)
                a[j] = a[j-h];

            a[j] = v;
        }
    } while (h != 1);
}
```

Note that when $h == 1$, the algorithm makes a pass over the entire list, comparing adjacent elements, but doing very few element exchanges. For $h == 1$, shell sort works just like insertion sort, except the number of inversions that have to be eliminated is greatly reduced by the previous steps of the algorithm with $h > 1$.

8/5/00

7 of 8

Shell Sort Example

8/5/00

6 of 8

Insertion Sort vs Shell Sort vs ??

The to shell sort is to find an optimal increment sequence that for any randomly selected input array, provides a subquadratic running time. The key is to eliminate inversions as quickly as possible doing the minimal number of comparisons.

A poorly selected increment sequence can lead to quadratic running time. This is the case of insertion sort, which selects a increment of 1 to start with.

The author of the course text books provides a different increment sequence that the one I have shown here. Try changing the shellSort implementation to use a different increment, and run some sample cases to see what the difference is for increasing size of n.

Shell sort is the algorithm of choice when you want a reasonably fast algorithm that is easy to implement. So, in your List<T> template class, the sorting algorithm you can provide in the implementation is shell sort as it will work well for linked lists of upto ~250,000 nodes, which is a very large linked list. When you start dealing with lists larger than this, it is usually better to switch to an external sorting algorithm that sorts a file-based list.

A slightly more complicated algorithm, but one which is $O(n \lg n)$ on average is QuickSort. QuickSort exploits the same idea of sorting sublists as does shell sort. Shell sort does an iterative processing of the entire list, which processes each of the sublists in the process. QuickSort uses a "divide and conquer" strategy, which divides the original list into two sublists around a pivot point, and then recursively sorts each sublist, merging them back together at the end of the sort.

8/5/00

8 of 8