Searching & Sorting in Java – Shell Sort Design & Implementation

The sort is based upon the following idea: Rather than sorting the entire list at once, we sort every k^{th} element. Such a list is said to be *k-sorted*. A k-sorted list is made up of *k* sublists, each of which is sorted, interleaved together.

The shellSort is supposed to start at a high value of k, and work towards a smaller value. Once the array has been 1-sorted, the sort is complete.

This description allows us to assemble a descriptive design for the algorithm.

```
shellSort(list)
        determine starting k-value for current list
        while k >= 1
                kSort the list using the current k-value
                determine the new k-value for the list
```

This design is very general, but it also highlights where most of the effort is required for the design to move forward.

1. determine the starting k-value
2. kSort the list using the k-value
3. determine the new k-value

For now, we will focus on the kSort itself. This algorithm needs to traverse the list multiple times, since each traversal will only look at every kth element. In other words, if we are inspecting every kth element, then we must traverse the list k times to see every element.

```
kSort(list, k)
        for start = 0 to k-1
                kSortSublist(list, k, start)
```

The kSortSublist method is only responsible for a single pass through the list, but is uses the same approach as insertion sort. It will sort every kth element in the list.

Consider the basic algorithm for insertionSort:

```
for (int top = 1; top < list.length; top++)
      // insert element found at top into its correct position
      // among the elements from 0 to top - 1
```

Let us examine each part of this algorithm, one piece at a time:

1. The first element in the array is at position 0. Our first pass sorts the first two elements (from 0 to top, which is set to 1). For a kSortSublist, the first position of our array is start, so our first pass sorts the two elements at start and start+k.

2. The insertionSort ends when we have reached (or go past) the end of the array. This condition still applies for our kSortSublist.

3. Since we sort every kth element, we need to increase top by k, rather than 1, for each iteration.

4. The body of the kSortSublist should insert the element at top into its correct position among the elements from start to top-k, only considering every kth element.

```
for (int top = start + k; top < list.length; top = top + k)
        // insert element at top into its correct position
        // among the elements from start to top-k,
        // only considering every kth element
```

Implemented in Java, we might code this algorithm as follows (once again, using insertionSort as our starting point).

```
public static void kSortSublist(int[] list, k, start)
{
      for (int top = start + k; top < list.length; top = top + k)
      {
            int item = list[top];       // temporary storage of item
            int i = top;

            while (i > 0 && item < list[i-k])
            {
                  // shift larger items to the right by k
                  list[i] = list[i-k];
                  // prepare to check the next item, k spaces left
                  i = i - k;
            }
            list[i] = item;        // put sorted item in open location
      }
}
```