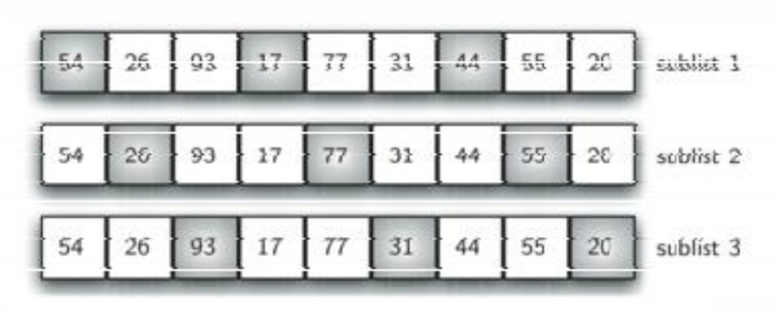
Prasad Institute of Technology, Jaunpur
Department of Computer Science & Engineering

Design and Analysis of Algorithm
BCS 503: Session 2025-26

The Shell Sort

The **shell sort**, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i, sometimes called the **gap**, to create a sublist by choosing all items that are i items apart.

This can be seen in given figure−1 This list has nine items. If we use an increment of three, there are three sublists, each of which can be sorted by an insertion sort. After completing these sorts, we get the list shown in Figure 2. Although this list is not completely sorted, something very interesting has happened. By sorting the sublists, we have moved the items closer to where they actually belong.



Figure−1: A Shell Sort with Increments of Three

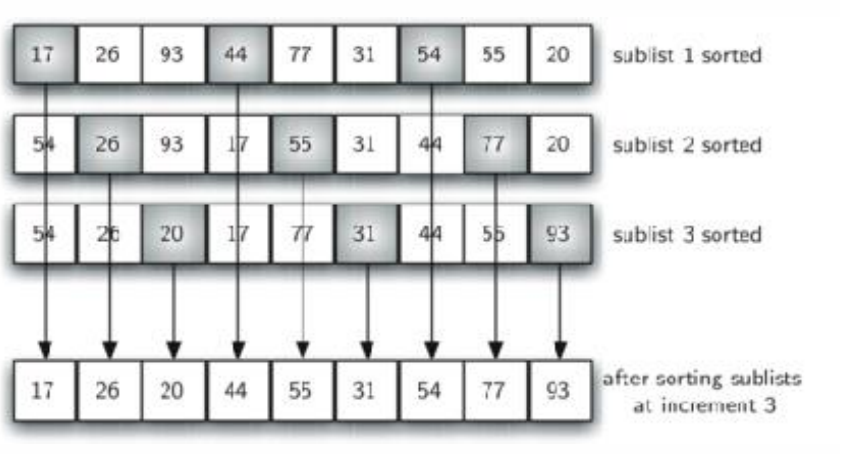


Figure 2: A Shell Sort after Sorting Each Sublist

Design and Analysis of Algorithm
BCS 503: Session 2025-26

Figure 3 shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.

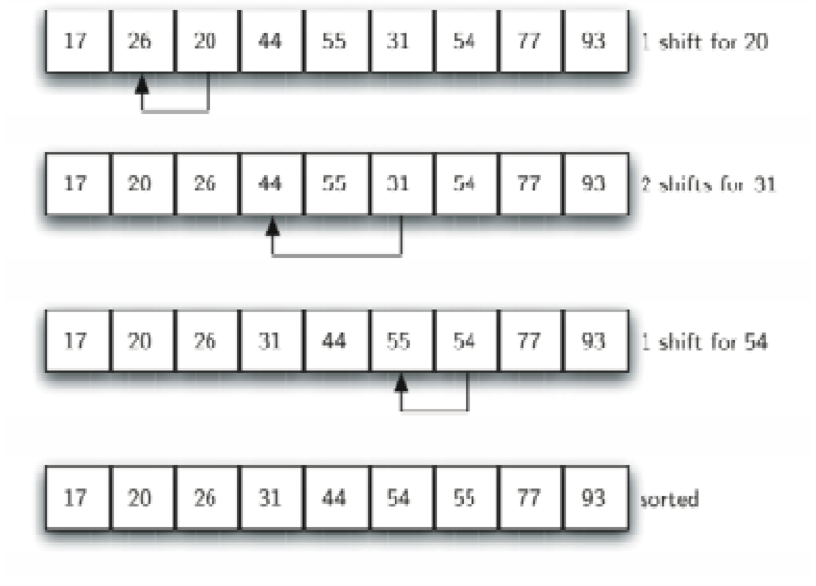


Figure 3: Shell Sort: A Final Insertion Sort with Increment of 1

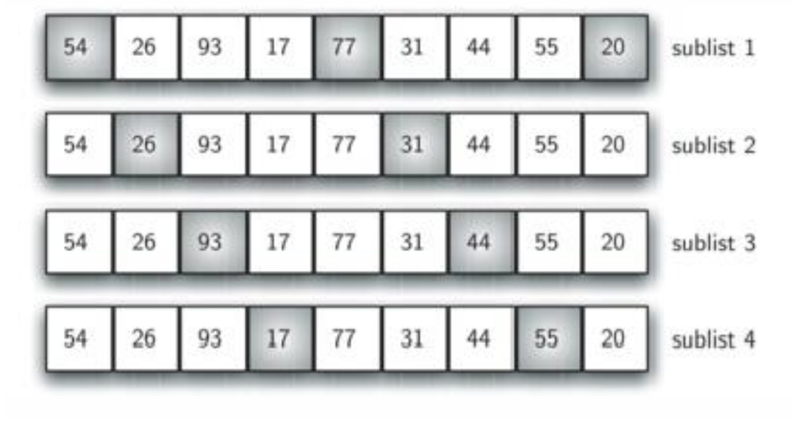


Figure 4: Initial Sublists for a Shell Sort

We said earlier that the way in which the increments are chosen is the unique feature of the shell sort. In this case, we begin with n/2 sublists. On the next pass, n/4 sublists are sorted. Eventually, a single list is sorted with the basic insertion sort. Figure 4 shows the first sublists for our example using this increment.

```
int shellSort(int arr[], int n)
{
   // Start with a big gap, then reduce the gap
   for (int gap = n/2; gap > 0; gap /= 2)
   {
      // Do a gapped insertion sort for this gap size.
      // The first gap elements a[0..gap−1] are already in gapped order
      // keep adding one more element until the entire array is
      // gap sorted
      for (int i = gap; i < n; i += 1)
      {
         // add a[i] to the elements that have been gap sorted
         // save a[i] in temp and make a hole at position i
         int temp = arr[i];

         // shift earlier gap−sorted elements up until the correct
         // location for a[i] is found
         int j;
         for (j = i; j >= gap && arr[j − gap] > temp; j −= gap)
            arr[j] = arr[j − gap];

         //  put temp (the original a[i]) in its correct location
         arr[j] = temp;
      }
   }
   return 0;
}
```

Analysis of Shell Sort

At first glance you may think that a shell sort cannot be better than an insertion sort, since it does a complete insertion sort as the last step. It turns out, however, that this final insertion sort does not need to do very many comparisons (or shifts) since the list has been pre−sorted by earlier incremental insertion sorts, as described above. In other words, each pass produces a list that is "more sorted" than the previous one. This makes the final pass very efficient.

We can say that it tends to fall somewhere between O(n) and O(n^2) based on the behavior Described above