Searching & Sorting in Java – Shell Sort

Although our previous sorting algorithms (insertion, selection, bubble) work well enough for short lists, they tend to slow down drastically with long lists. This is due to the large number of comparisons that are needed to find the correct positions for each data element.

This is not always true. If, for example, the data is almost in order, an algorithm like insertion sort will be very fast, because only a few comparisons are needed for each value.

Shell sort adapts insertion sort to produce a sorting technique that works well even with long lists that are randomly ordered. The key is that data can be moved long distances in the list with only a few comparisons.

The sort is based upon the following idea: Rather than sorting the entire list at once, we sort every kth 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.

Suppose we are given an unsorted list of integers.

34	21	40	12	27	18	29	13	25	17	11	38

This can be decomposed into three sublists, each with 4 elements.

34			12			29			17		
	21			27			13			11	
		40			18			25			38

If we now sort each sublist independently, we obtain

12			17			29			34		
	11			13			21			27	
		18			25			38			40

which yields the following 3-sorted list.

12	11	18	17	13	25	29	21	38	34	27	40

If we compare the unsorted list, to the 3-sorted list, to the sorted list, we see that the 3-sorted list has moved significantly toward the sorted goal by sorting small sublists (which is comparatively fast).

Unsorted	34	21	40	12	27	18	29	13	25	17	11	38
3-sorted	12	11	18	17	13	25	29	21	38	34	27	40
Sorted	11	12	13	17	18	21	25	27	29	34	38	40

Shell sort repeatedly used insertion sort to create sorted sublists. Initially a *k*-sorted list is created using a large value of *k* so that values can be moved long distances. On subsequent passes, smaller values of *k* are used, until, on the final pass, the value of *k* is set to one. A 1-sorted list is completely sorted.

Searching & Sorting in Java – Shell Sort

Consider the following example which creates lists that are 7-sorted, 3-sorted, and then 1-sorted.

Unsorted	64	31	10	40	22	49	82	20	29	56	40	18	19	27	26
	20							26							64
		29							31						
			10							56					
First Pass				40							40				
					18							22			
						19							49		
							27							82	
7-sorted	20	29	10	40	18	19	27	26	31	56	40	22	49	82	64
	20			27			40			49			56		
Second Pass		18			26			29			40			82	
			10			19			22			31			64
3-sorted	20	18	10	27	26	19	40	29	22	49	40	31	56	82	64
Third Pass 1-sorted	10	18	19	20	22	26	27	29	31	40	40	49	56	64	82

In using Shell Sort, we noted that the first pass should used a large value of *k* so that values could move long distances quickly. We also noted that the last pass must use *k* = 1 to completely sort the list. This discussion begs the question: What is the best sequence of values for *k*? The answer to that question is not well defined, despite a considerable amount of work done on the problem.

One sequence that has been found to give good results follows the pattern 1, 4, 13, 40, ...

Each term in the sequence is three times the previous term, plus one.

Searching & Sorting in Java – Shell Sort

Exercises

1. Given the following data, show how they would appear after they have been 5-sorted.

   26 37 21 41 63 19 61 72 55 29 47 18 26 22

2. Starting with the same set of data from question 1, show how they would appear 4-sorted.

3. How would you answer the following argument against using Shell Sort? "The last step of Shell Sort, using *k* = 1, is simply a normal insertion sort. Since Shell Sort performs many preliminary steps before this final one, it must be slower than a single insertion sort."

4. Suppose you were going to write a version of Shell Sort using the sequence of k-sorts suggested previously. For a list containing *n* elements, the first value of *k* that should be used is the largest value in the sequence that is less than *n*. For example, in a list of 50 elements, the largest *k* value would be 40, so the first pass should be 40-sorted.

   (a) Write a sequence of statements that will initialize *k* correctly for a given value of *n*.

   (b) Write a statement that will, for any value of *k* in the sequence, produce the next smaller value of *k*.

5.
   (a) Write a method `shellSort` to sort an array of `int` values in ascending order. In performing the *k*-sorts, use the sequence of values of *k* suggested in the text. Be sure to use insertion sort at each stage of the sort.

   (b) Test your method by writing a complete program that first generates an array of 500 random `int` values in the range [0, 999], prints this array (ten values per line), sorts the array using your `shellSort` method, and then prints the resulting array (ten values per line).

6. Experiment with using Shell Sort with sequences where *k* is other than the suggested values, testing your sequences on large arrays of integers and noting the time required by the sort in each case. To measure the time taken by a sort, you can use the method `currentTimeMillis()` in the `System` class. The method has the signature

   ```
   public static long currentTimeMillis()
   ```

   and it returns a long value, the number of milliseconds since midnight, January 1, 1970.

Searching & Sorting in Java – Shell Sort

Solutions

1. Given the following data, show how they would appear after they have been 5-sorted.

   26 37 21 41 63 19 61 72 55 29 47 18 26 22

   Broken into every 5th element:

26					19					47			
	37					61					18		
		21					72					26	
			41					55					22
				63					29				

   Sorted sub-lists:

19					26					47			
	18					37					61		
		21					26					72	
			22					41					55
				29					63				

   5-sorted list:

   19 18 21 22 29 26 37 26 41 63 47 61 72 55

2. Starting with the same set of data from question 1, show how they would appear 4-sorted.

   4-sorted list:

   26 19 21 41 26 22 47 72 55 29 61 18 63 37

3. How would you answer the following argument against using Shell Sort? "The last step of Shell Sort, using *k* = 1, is simply a normal insertion sort. Since Shell Sort performs many preliminary steps before this final one, it must be slower than a single insertion sort."

   The goal of the previous steps is to create a partially sorted list, so that when we reach the single insertion sort, it will proceed very quickly. If we didn't perform the previous k-sorts, the single insertion sort would take much longer (for typical, random data).

4. Suppose you were going to write a version of Shell Sort using the sequence of k-sorts suggested previously. For a list containing *n* elements, the first value of *k* that should be used is the largest value in the sequence that is less than *n*. For example, in a list of 50 elements, the largest *k* value would be 40, so the first pass should be 40-sorted.

a) Write a sequence of statements that will initialize *k* correctly for a given value of *n*.

The formula for the value of k can be expressed as: $k_n = 3 \times k_{n-1} + 1$

In Java, this can be expressed as: k = 3 * previousK – 1

The entire code fragment in Java (which should probably be put into a method):

```
int k = 1;   // 1 is the default start for k
int prevK;
while (k < n)
{
   prevK = k;   // keep track of previous valid value of k
   k = 3 * prevK + 1
}
// the current value of k is actually larger than n,
// so use prevK
return prevK;
```

b) Write a statement that will, for any value of *k* in the sequence, produce the next smaller value of *k*.

Given $k_n = 3 \times k_{n-1} + 1$, we can rearrange to $k_{n-1} = \frac{k_n + 1}{3}$.

5.