**Algorithm Study –Insertion Sort**

**Algorithms**: Insertion sort

**aka**: Linear Insertion sort

**Techniques**: Insertion sort uses in-place sorting

**Categories**: Sorting

**Problem**: The Insertion sort is designed to be efficient for sorting small datasets, especially those which are already partially sorted.

**Applications**: The Insertion sort is often used in situations where the data being sorted is nearly sorted already, and few swaps are anticipated. Most applications are geared toward small datasets.

**References**:

* <http://xlinux.nist.gov/dads/HTML/insertionSort.html>
* <http://rosettacode.org/wiki/Sorting_algorithms/Insertion_sort>
* *Algorithms*, 4th edition, by Sedgewick & Wayne (pages 250-252)

**Implementation details**:

* **Big Idea**: The list is iterated through, starting with the second value, inserting one value where it belongs in the sorted list each time, with respect to values already sorted, until the list is completely sorted.
* **Description**: For each value in the list, Insertion sort checks to see whether the preceding value is larger or smaller. If smaller, the sort continues on to the next value. If larger, it looks farther back in the list to find the correct position and inserts the smaller value there, shifting the other values forward to compensate. As Insertion sort iterates through the list, the values behind the one currently being evaluated become sorted.
* **Pseudo-code**: (adapted from <http://rosettacode.org/wiki/Sorting_algorithms/Insertion_sort>)

**function** *insertionSort*(array A)

//Iterate through entire list

**for** i **from** 0 **to** length[A] **do**

//While not at start of list and preceding value

//is larger

**while** j > 0 **and** A[j] < A[j-1] **do**

//If preceding value higher, swap

A[j+1] := A[j]

j := j-1

**done**

//Advance to next value

A[j+1] = value

**done**

* **Specific implementation**: (see ShellSortImp.java, it has both sorts from this report)

**Correctness**:

**Theoretical**: This sort is widely used and well-understood. Picking a value next to those already sorted and inserting it into the correct place among the sorted values does, over several iterations, yield a sorted list.

**Empirical**: (as seen in ShellSortImp.java) I tested with ten, one hundred, and one thousand values. Each test resulted in a sorted array, a sampling of which is output by the program for visual verification.

**Performance**:

**Theoretical**: Big O is O(n2)

**Empirical**: (as seen in ShellSortImp.java) I tested with ten, one hundred, and one thousand values.

Ten values:

* Completed in 0.003 milliseconds
* Used 25 swaps

One hundred values:

* Completed in 0.160 milliseconds
* Used 2513 swaps

One thousand values:

* Completed in 10.546 milliseconds
* Used 255,954 swaps

\*(note that these values differ from one run to the next due to the use of random integers to fill unsorted arrays)

**Anecdotes**: <none>

**History**: I was unable to learn who created the Insertion sort, but it was improved upon by D.L. Shell, who created the Shell sort in the late 1950’s.

**Variations**: Shell sort is a variation of Insertion sort.

**Alternatives**: In professional programming, the Quicksort is usually favored over the Insertion sort, though it isn’t always a better solution.

**Credits:**

* <http://en.wikipedia.org/wiki/Insertion_sort>
* <http://stackoverflow.com/questions/14619504/shell-sort-and-insertion-sort>
* <http://www.stoimen.com/blog/2012/02/13/computer-algorithms-insertion-sort/>

**Algorithm Study – Shell Sort**

**Algorithm**: Shell sort

**aka**: Shell’s method

**Techniques**: Shell sort uses a diminishing increment sequence

**Categories**: Sorting

**Problem**: The Shell sort was developed as an improvement of the Insertion sort, and is designed to use fewer swaps.

**Applications**: The Shell sort is useful in the same situations where one would normally employ an Insertion sort, with an advantage due to fewer swaps being made.

**References**:

* <http://java.dzone.com/articles/algorithm-week-shell-sort>
* <http://xlinux.nist.gov/dads/HTML/shellsort.html>
* <http://rosettacode.org/wiki/Sorting_algorithms/Shell_sort>
* <http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Shell_sort.html>
* *Algorithms*, 4th edition, by Sedgewick & Wayne (pages 258-262)

**Implementation details**:

* **Big Idea**: Shell sort is a generalization of Insertion sort that allows swapping list values which are far apart.
* **Description**: Shell sort can be described as a sequence of insertion sorts that is based on a diminishing increment sequence. There are many of these sequences, but I’ve chosen to use Shell’s sequence (the original). Shell’s sequence follows the formula FLOOR(N/2k). So, if N = 1000, the sequence would be 500, 250, 125, and so on. This sequence, starting with (N/2), or 500 in my example, determines the gap between elements being compared. The gap is reduced by half each pass. Eventually, the gap is reduced down to 1, which is the traditional Insertion sort. The ability to swap values which are far away generally reduces the number of swaps performed, as compared to Insertion sort.
* **Pseudo-code**: (adapted from <http://en.wikipedia.org/wiki/Shellsort>)

*# Sort an array a[0...n-1].*

length = a.length

gap = floor(length/2)

foreach (gap **in** gaps)

{

*# Do an insertion sort for each gap size.*

**for** (i = gap; i < length; i += 1)

{

temp = a[i]

j = i

**while** (j >= gap **and** a[j - gap] > temp)

{

a[j] = a[j - gap]

j -= gap

}

a[j] = temp

}

gap = floor(gap/2)

}

* **Specific implementation**: (see ShellSortImp.java, it has both sorts from this report)

**Correctness**:

**Theoretical**: The diminishing increment sequence of Shell sort does eventually result in the use of an Insertion sort, which is capable of sorting the list on its own.

**Empirical**: (as seen in ShellSortImp.java) I tested with ten, one hundred, and one thousand values. Each test resulted in a sorted array, a sampling of which is output by the program for visual verification.

**Performance**:

**Theoretical**: Big O is O(n2) for Shell’s sequence. Other sequences perform differently.

**Empirical**: (as seen in ShellSortImp.java) I tested with ten, one hundred, and one thousand values.

Ten values:

* Completed in 0.423 milliseconds
* Used 11 swaps

One hundred values:

* Completed in 0.051 milliseconds
* Used 429 swaps

One thousand values:

* Completed in 0.819 milliseconds
* Used 7805 swaps

\*(note that these values differ from one run to the next due to the use of random integers to fill unsorted arrays)

**Anecdotes**: <none>

**History**: D.L. Shell developed the Shell sort as an improvement of the Insertion sort in the late 1950’s. Other individuals such as Knuth and Pratt have proposed different diminishing increment sequences for the sort, but the superiority of any sequence with the sort depends on the application.

**Variations**: Shell sort is a variation of Insertion sort.

**Alternatives**: In professional programming, the Quicksort is usually favored over Shell sort, though it isn’t always a better solution.

**Credits:**

* <http://en.wikipedia.org/wiki/Shellsort>
* <http://stackoverflow.com/questions/14619504/shell-sort-and-insertion-sort>
* <http://www.stoimen.com/blog/2012/02/27/computer-algorithms-shell-sort/>

**Insertion Sort and Shell Sort Compared**

Both sorts, used on identical data, resulted in identical sorted arrays. Although Shell sort always used fewer swaps, it didn’t always run faster than Insertion sort. This was likely due to variations in test data, with the input arrays being filled by random integers.

Shell sort is generally favored over Insertion sort, especially for use with large datasets. This is because its strategy of starting by sorting values that are far apart yields a partially sorted list that its eventual Insertion sort (when the increment reaches one) quickly handles. An Insertion sort is slower because it can only move through the array from one side to the other, one place at a time. This limits the effectiveness of the Insertion sort to small and perhaps moderately sized datasets. The Shell sort, however, can be used for large datasets without a huge loss in running time or a devastatingly high number of swaps.

However, as previously stated, the Quicksort is often favored over both the Insertion and Shell sorts due to its (sometimes only perceived) superiority. Following this study, I would recommend Insertion sorts for only datasets of one hundred values or fewer. For datasets between one hundred and one thousand values, I would choose Shell sort, and for more than one thousand values, Quicksort is most likely best.