Sorting introduction

Rearrange an array of n items in ascending order according to a defined key

Sort can compare of data of different types with something called a callback

**Callback**: a reference to executable code

* Client passes array of objects to sort() function
* The sort() function calls back object’s compareTo() method (associated with objects in the array) as needed

Callback implementation:

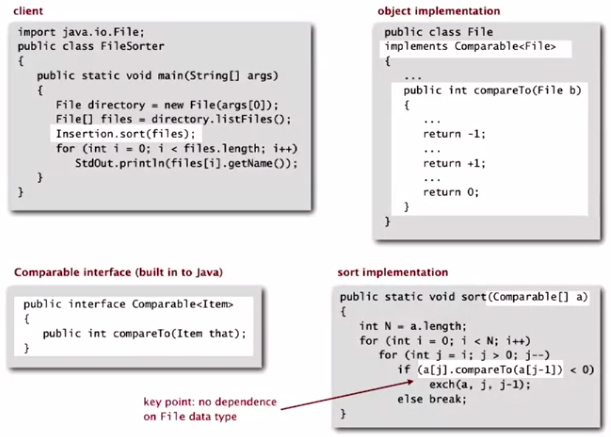
* **Java:** interfaces.
* **C:** function pointers
* **C++:** class-type functors
* **C#:** delegates
* **Python, Perl, ML, Javascript:** first class functions

Java **interfaces**:

**Comparable interface:** data types that implement Comparable will have a compareTo() method

Sorted objects will need to implement Comparable. compareTo returns -1 for less than, +1 for greater than and 0 for equal to. We simply use this compareTo() in the sort method.

Array of the type Comparable[] as a parameter. The sort code uses the compareTo() method to test whether one instance is < > or = another.



Total order is required for sorts. ITEMS MUST BE ABLE TO BE PUT IN AN ORDER.

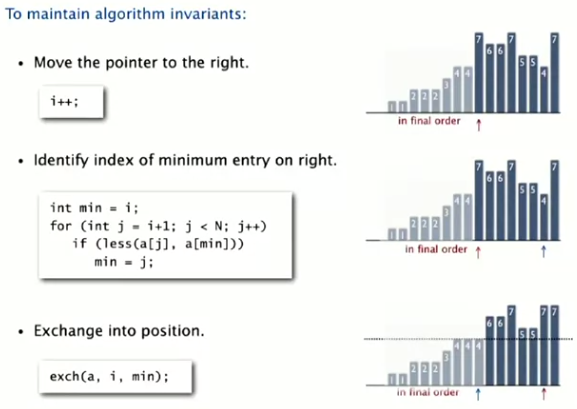
The Comparable API in Java (compareTo() can work as v.compareTo(w) )

* Is a total order
* Returns a negative integer, zero or positive integer
* Throws an exception in case of incompatible types (or either value is null)

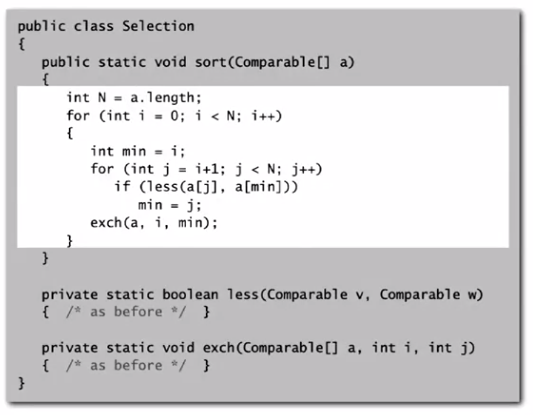
Selection Sort

In iteration *I* find index *min* of smallest remaining entry, swap a[i] and a[min]

Within the algorithm the pointer scans from left to right only. Nothing on the left of the pointer is larger than anything to the right of the pointer.



Selection sort Java implementation:

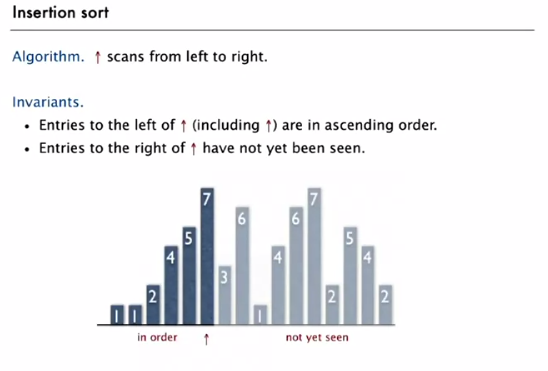


Selection sort is insensitive to input order: It always goes through entire array for the minimum, so it will be quadratic time.

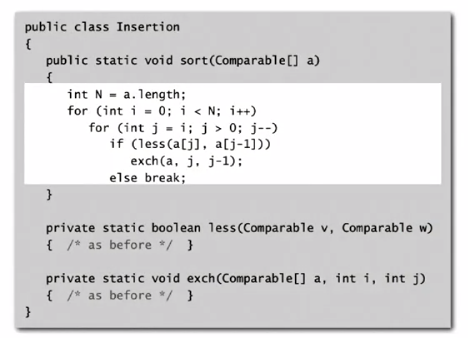
Data movement is minimal: numbers are only exchanged into their final position with one exchange.

Insertion Sort

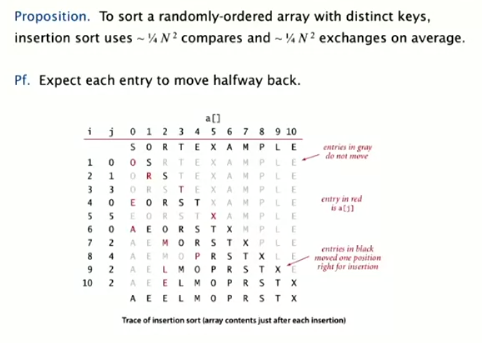
Elements to the left of the pointer are in ascending order, but elements to the right are unknown and may be less than any item on the left of the pointer.



Once you move the pointer to the right, you must exchange that new element with every object on its left that is greater



Visual mathematical analysis



Best case for insertion sort: ascending order == no exchanges and just n-1 compares

Worst case for insertion sort: descending order == every element goes back. ½ N^2 compares and ½ N^2 exchanges.

Therefore, insertion sort is faster than selection sort in the best case and slower in the worst case.

When to use insertion sort: partially sorted arrays. These are common, such as with arrays that have just had some unsorted entries added to the end, but everything else is sorted. An array is partially sorted if number of inversions is less than or equal to some constant times N (linear).

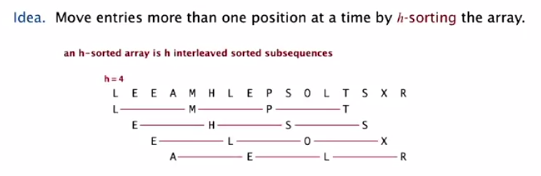
Insertion sort is linear time for partially sorted arrays.

Shellsort

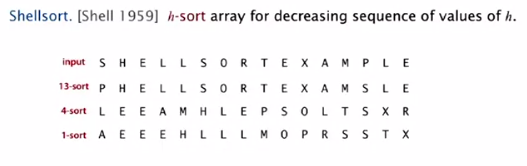
Insertion sort is inefficient because elements only move one position at a time even if we know they have a long way to move.

Shellsort moves several positions at a time thanks to ‘h-sorting’

H-sorting is when you have multiple interleaved subsequences that are sorted



Looking at every fourth element above you will notice that the ‘shell’ of elements is sorted.



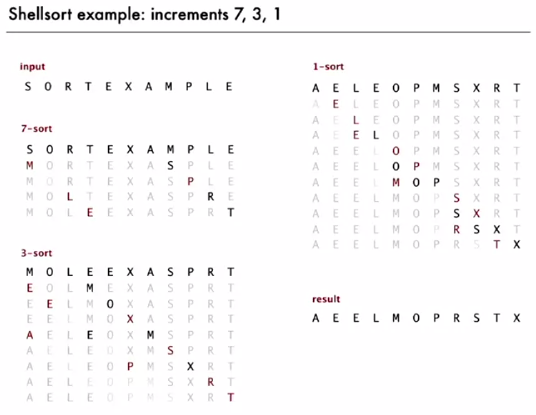
How do you h-sort?

Insertion sort, but move h-back, not just 1.

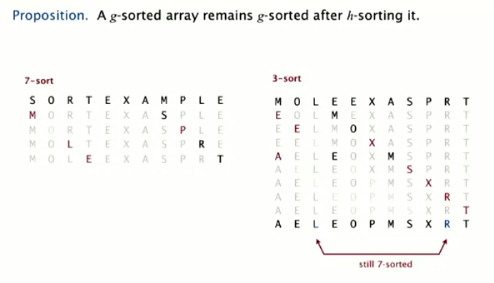


Why?

* Insertion sort is good because large increments == small subarrays to sort
* When increments are small, they are nearly in order (because of previous h-sorts with larger values of h). Insertion sort is fast with partially sorted arrays



Shellsort is more efficient because the its increments are not effected when they become smaller.



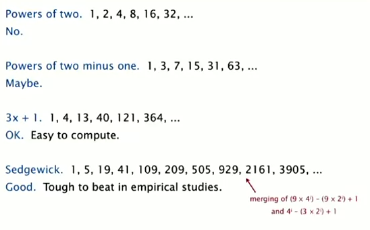
Increment sequences for shellsort:

NO: Powers of 2. Why? Does not compare even to odd positions until 1-sort.

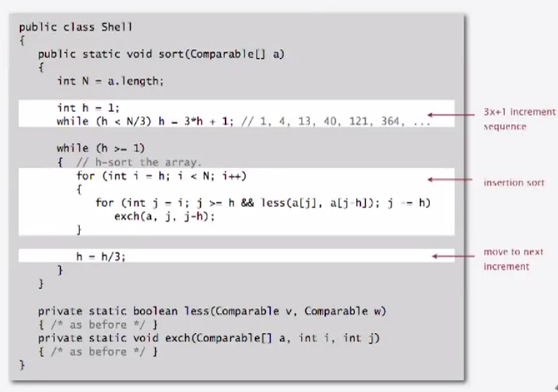
Maybe: Powers of 2 -1. (1, 3, 7, 15, 32, 63)

Okay: 3x + 1. (1, 4, 13, 40, 121, 364)

Good: see below:



Shellsort Java implementation:



Worst time for this is O(N^3/2)

When is shellsort so good?

It’s fast excepting very large arrays

The code is very small and efficient, so it’s good for embedded systems

Shuffling

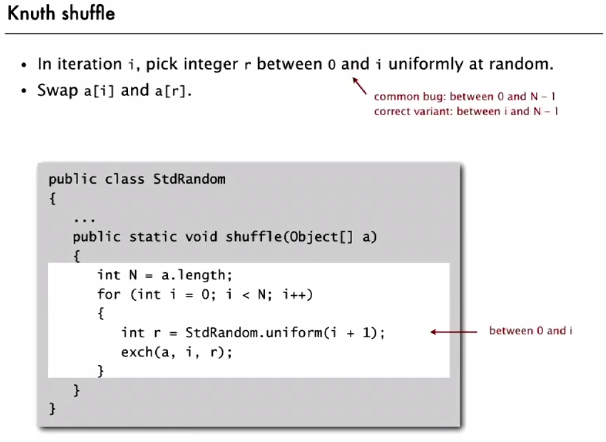
One option is to generate a real number for each item in the array, then sort by real numbers, but that takes the time of a sort to do.

The better option:

LINEAR TIME -> Knuth shuffle

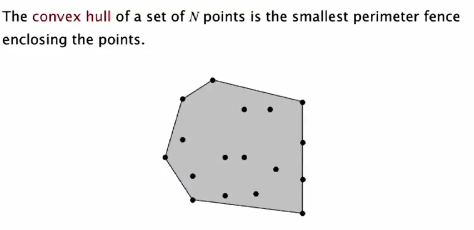
*I* starts at 0. Swap with a uniformly random int between 0 and *I-1.* Then increment *I* and repeat.

Java implementation:



Uniformly random number should be between *I* and *I*-1 (not 0 to N – 1). Alternatively, can do *I* to N -1 as an alternative.

Convex Hull



Mechanical algorithm is to hammer nails into points and tie a string around the border

