# **insertion sort**

Insertion sort is a simple sorting algorithm, it builds the final sorted array one item at a time. It is much less efficient on large lists than other sort algorithms.

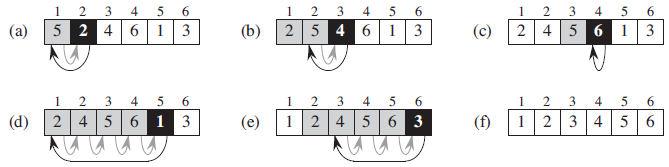
Advantages of Insertion Sort:

1) It is very simple.

2) It is very efficient for small data sets.

3) In-place; i.e., only requires a constant amount O(1) of additional memory space.

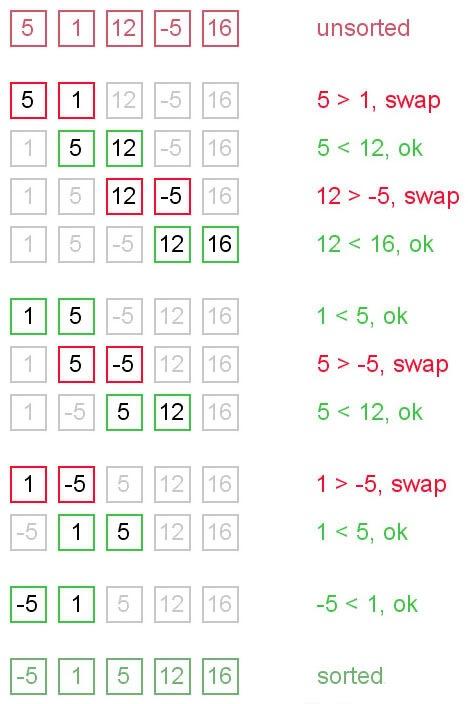
Insertion sort iterates through the list by consuming one input element at each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.



The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., Θ(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array. The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n2)). The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

# **bubble sort**

Bubble sort, also referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists.

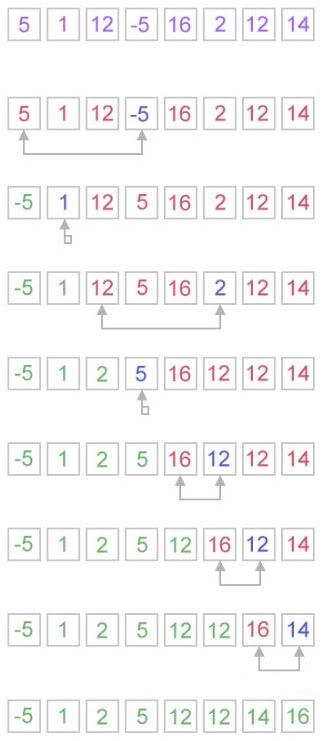


Bubble sort has worst-case and average complexity both О(n2), where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of O(n log n). Even other О(n2) sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large. Performance of bubble sort over an already-sorted list (best-case) is O(n).

# 

# **selection sort**

The selection sort is a combination of searching and sorting. During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array. The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.



Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takesn − 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n − 1 elements and so on, for (n − 1) + (n − 2) + ... + 2 + 1 = n(n − 1) / 2 ∈ Θ(n2) comparisons. Each of these scans requires one swap for n − 1 elements.

Exercise:

Speed test 3 different sorting methods(selection sort, bubble sort, insertion sort) to compare(Sorting methods are in Sort.java). Do best case (already sorted), worst case( reverse sorted), and average case(average of about 20 arrays) use 3 different array sizes: 100, 10000, and 100000. Use excel to draw a graph comparing these algorithms in best, worst, and average case.

Tips:

Basic code for timing a method:

long startTime = System.nanoTime();

methodToTime(); //sorting method you need to call in this case.

long endTime = System.nanoTime();

long duration = (endTime - startTime)/1000000; // time in milliseconds.

Generate arrays of different size:

Call **generateRandomArray()** method.

For example:

Integer data[];

data = generateRandomArray(100);

This gives you an array size of 100;