Contents

[Complexity analysis 2](#_Toc463259829)

[Insertion Sort 3](#_Toc463259830)

[Selection Sort 5](#_Toc463259831)

[Heap Sort 7](#_Toc463259832)

[Quick Sort 9](#_Toc463259833)

[Radix Sort 12](#_Toc463259834)

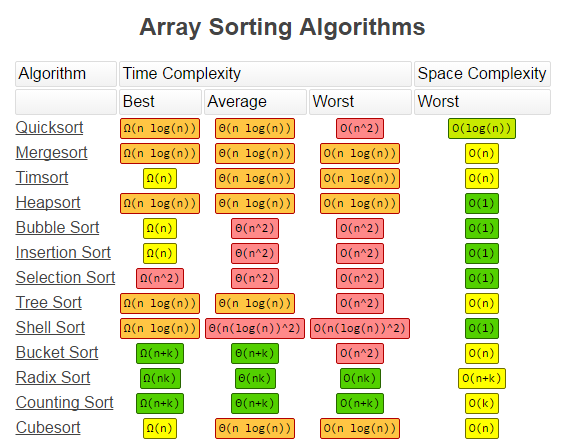
[Counting Sort 14](#_Toc463259835)

[Bucket Sort 15](#_Toc463259836)

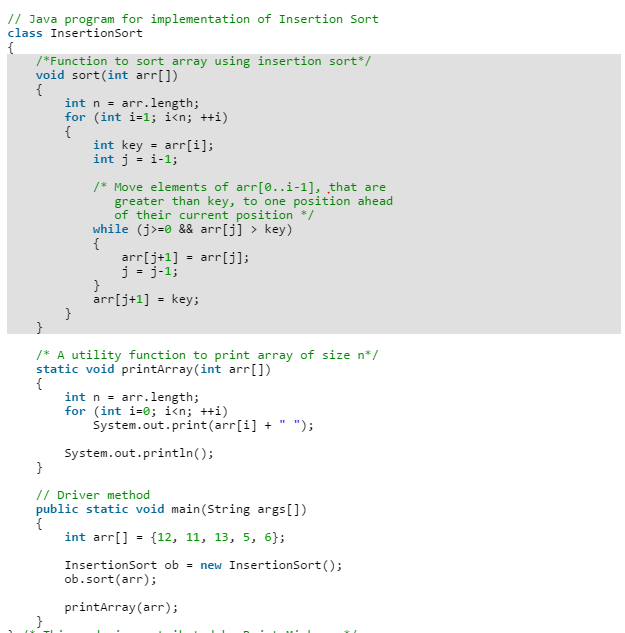
[Bubble Sort 16](#_Toc463259837)

[Merge Sort 18](#_Toc463259838)

# Complexity analysis



# Insertion Sort



Algorithm

// Sort an arr[] of size n  
insertionSort(arr, n)  
Loop from i = 1 to n-1.  
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

**Example:**  
**12**, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 5 (Size of input array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12  
**11, 12**, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13  
**11, 12, 13**, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.  
**5, 11, 12, 13**, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.  
**5, 6, 11, 12, 13**

**Boundary Cases**: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

**Algorithmic Paradigm:** Incremental Approach

**Sorting In Place:** Yes

**Stable:** Yes

**Online:** Yes

**Uses:** Insertion sort is uses when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

/\*\*

\*

\* Insertion Sort: Most Appropriate for small sets because stable and low memory overhead

\*

\* When N is guaranteed to be small, including as the base case of a quick sort or merge sort. While this is O(N^2), it has a very small constant and is a stable sort.

\*

\* Consider it as sorting two sets of values one to left and other to right.

\* We keep sorting elements to left by placing one element at a time from right to left in its

\* correct position.

\* [sorted]\^/[unsorted]

\* [4]\^/[9 8 3 9 1]

\* [4 9]\^/[8 3 9 1]

\* [4 8 9]\^/[3 9 1]

\* [3 4 8 9]\^/[9 1]

\* [3 4 8 9 9]\^/[1]

\* [1 3 4 8 9 9]\^/[]

\*

\* Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is O(nk) when each element in the input is no more than k places away from its sorted position

\* Stable; i.e., does not change the relative order of elements with equal keys

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

\* Online; i.e., can sort a list as it receives it

\*

\* Worst case performance О(n2) comparisons, swaps

\* Best case performance O(n) comparisons, O(1) swaps

\* Average case performance О(n2) comparisons, swaps

\* Worst case space complexity О(n) total, O(1) auxiliary

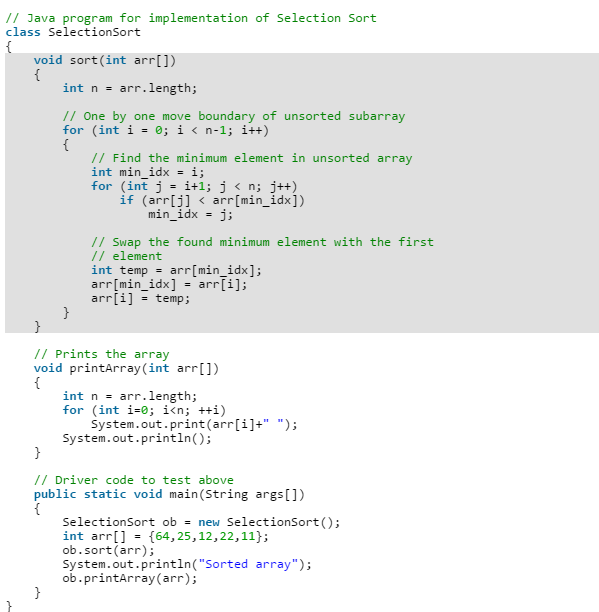
\*

\* **@author** XX61691

\*

\*/

# Selection Sort



\* Selection sort: is a sorting algorithm, specifically an in-place comparison

\* sort. It has O(n2) time complexity, making it inefficient on large lists, and

\* generally performs worse than the similar insertion sort. Selection sort is

\* noted for its simplicity, and it has performance advantages over more

\* complicated algorithms in certain situations, particularly where auxiliary

\* memory is limited.

\*

\* The algorithm divides the input list into two parts: the sublist of items

\* already sorted, which is built up from left to right at the front (left) of

\* the list, and the sublist of items remaining to be sorted that occupy the

\* rest of the list. Initially, the sorted sublist is empty and the unsorted

\* sublist is the entire input list. The algorithm proceeds by finding the

\* smallest (or largest, depending on sorting order) element in the unsorted

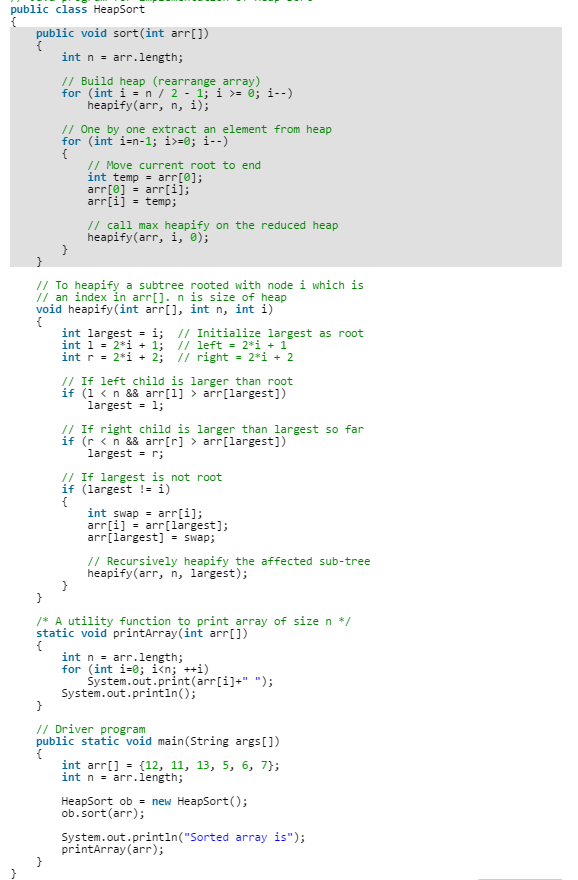
\* sublist, exchanging (swapping) it with the leftmost unsorted element (putting

\* it in sorted order), and moving the sublist boundaries one element to the

\* right.

The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

# Heap Sort



 A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source [Wikipedia](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees))

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Algorithm

**1.** Build a max heap from the input data.  
**2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.  
**3.** Repeat above steps until size of heap is greater than 1.

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

\* Heap sort: When you don't need a stable sort and you care more about worst

\* case performance than average case performance. It's guaranteed to be O(N log

\* N), and uses O(1) auxiliary space, meaning that you won't unexpectedly run

\* out of heap or stack space on very large inputs.

\*

\* Heap: Represent as Tree or Array.

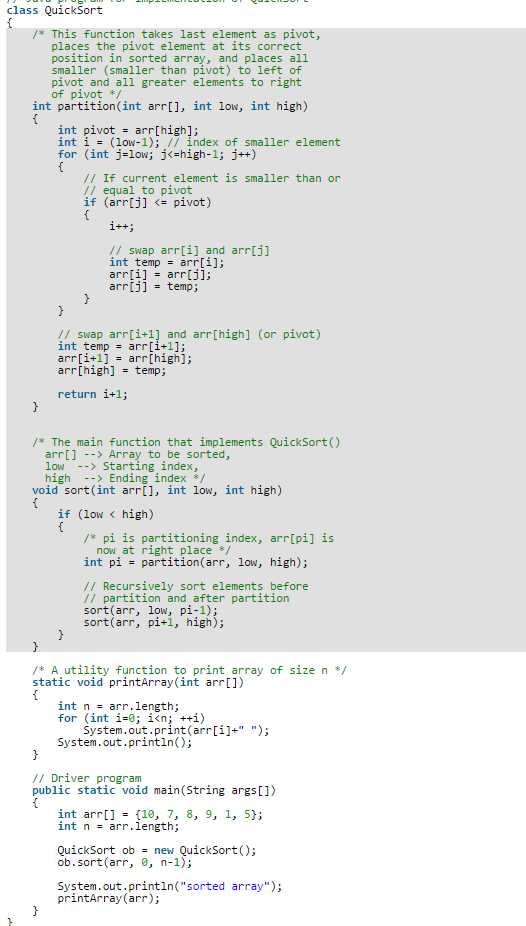
\* Array:

\* Child: |Parent:

\* n --->2n + 1 |(n-1)/2 <----n

\* !---->2n + 2 |

# Quick Sort



\* Quick sort: When you don't need a stable sort and average case performance

\* matters more than worst case performance. A quick sort is O(N log N) on

\* average, O(N^2) in the worst case. A good implementation uses O(log N)

\* auxiliary storage in the form of stack space for recursion.

\*

\*

\* 1. Pick an element, called a pivot, from the array. Picking last number as pivot is easier to implement :)

\* 2. Partitioning: reorder the array so that all elements with values less than the pivot

\* come before the pivot, while all elements with values greater than the pivot come

\* after it (equal values can go either way). After this partitioning, the pivot is

\* in its final position. This is called the partition operation.

\* 3. Recursively apply the above steps to the sub-array of elements with smaller values

\* and separately to the sub-array of elements with greater values.

\*

\* Worst case performance O(n2)

\* Best case performance O(n log n) (simple partition)

\* or O(n) (three-way partition and equal keys)

\* Average case performance O(n log n)

\* Worst case space complexity O(n) auxiliary (naive)

\* O(log n) auxiliary

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

**Why Quick Sort is preferred over MergeSort for sorting Arrays**  
Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

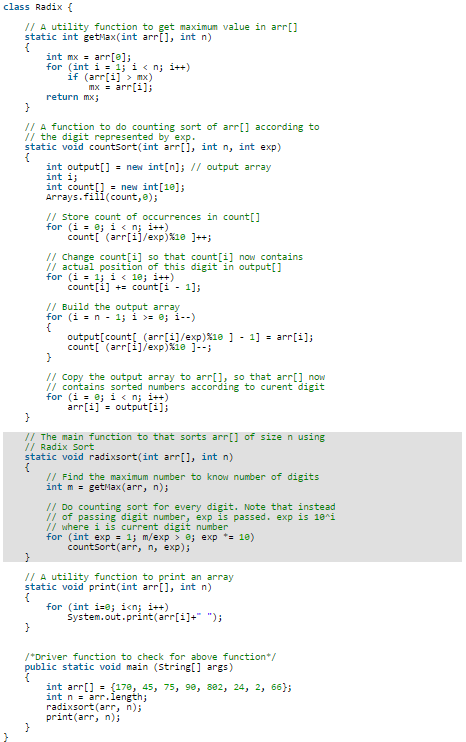
Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

**Why MergeSort is preferred over QuickSort for Linked Lists?**  
In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

# Radix Sort



**What if the elements are in range from 1 to n2?**  
We can’t use counting sort because counting sort will take O(n2) which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?  
[Radix Sort](http://en.wikipedia.org/wiki/Radix_sort) is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

***The Radix Sort Algorithm***  
**1)** Do following for each digit i where i varies from least significant digit to the most significant digit.  
………….**a)** Sort input array using counting sort (or any stable sort) according to the i’th digit.

**Example:**  
Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [\*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [\*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

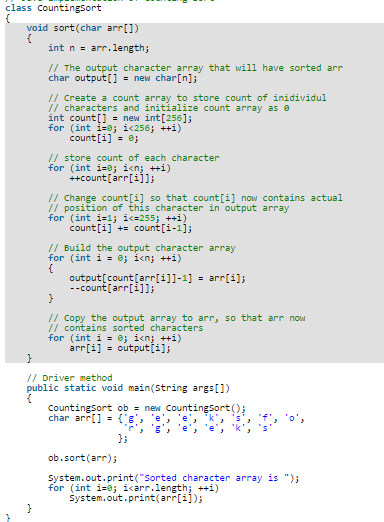
802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

# Counting Sort

[Counting sort](http://en.wikipedia.org/wiki/Counting_sort) is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.



# Bucket Sort

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.   
Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

**bucketSort(arr[], n)**

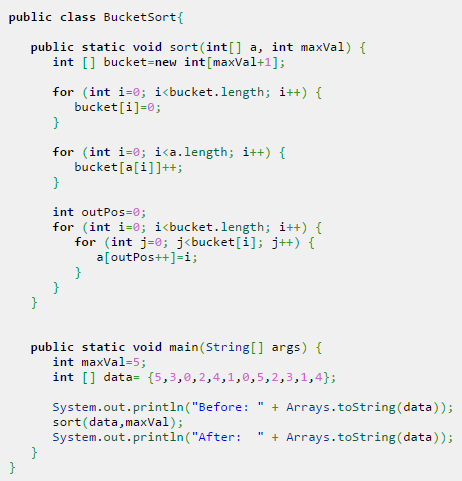
1) Create n empty buckets (Or lists).

2) Do following for every array element arr[i].

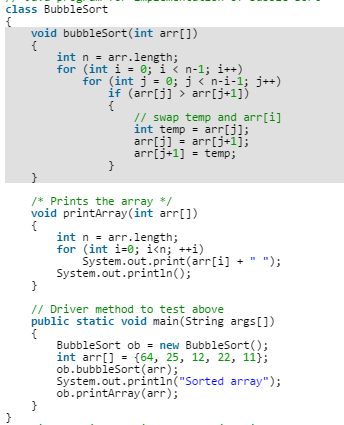
.......a) Insert arr[i] into bucket[n\*array[i]]

3) Sort individual buckets using insertion sort.

4) Concatenate all sorted buckets.



# Bubble Sort



[Bubble Sort](http://en.wikipedia.org/wiki/Bubble_sort)is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**Example:**  
**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one**whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

# Merge Sort

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

**Applications of Merge Sort**

1. [Merge Sort is useful for sorting linked lists in O(nLogn) time](http://www.geeksforgeeks.org/merge-sort-for-linked-list/).In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

1. [Inversion Count Problem](http://www.geeksforgeeks.org/counting-inversions/)
2. Used in [External Sorting](http://en.wikipedia.org/wiki/External_sorting)

**Auxiliary Space:** O(n)

**Algorithmic Paradigm:**Divide and Conquer

**Sorting In Place:** No in a typical implementation

**Stable:** Yes

