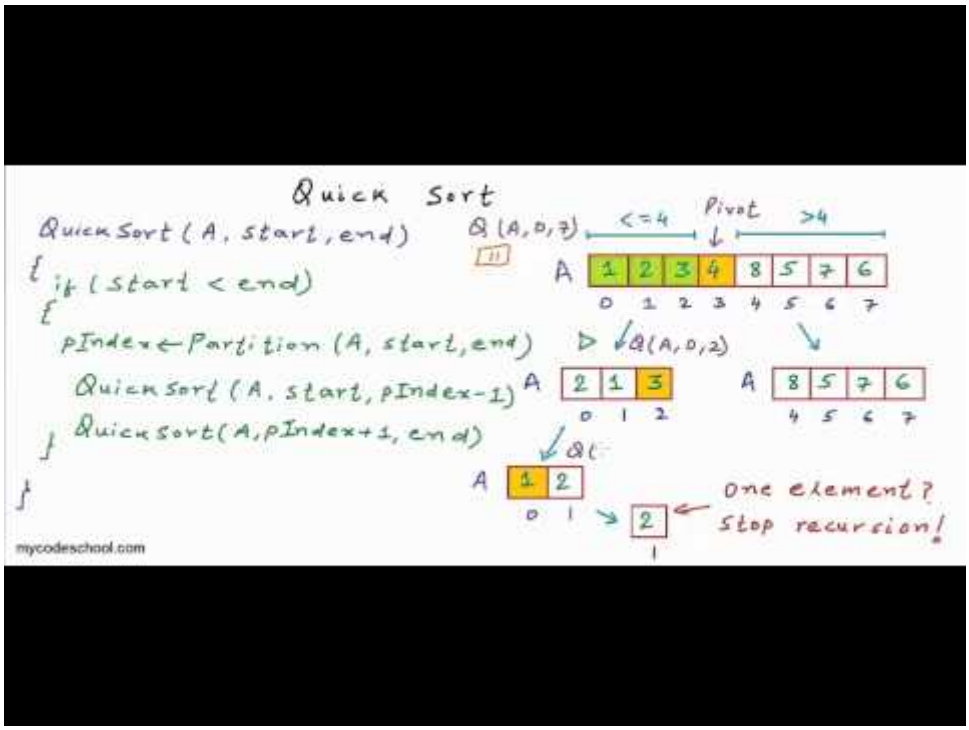
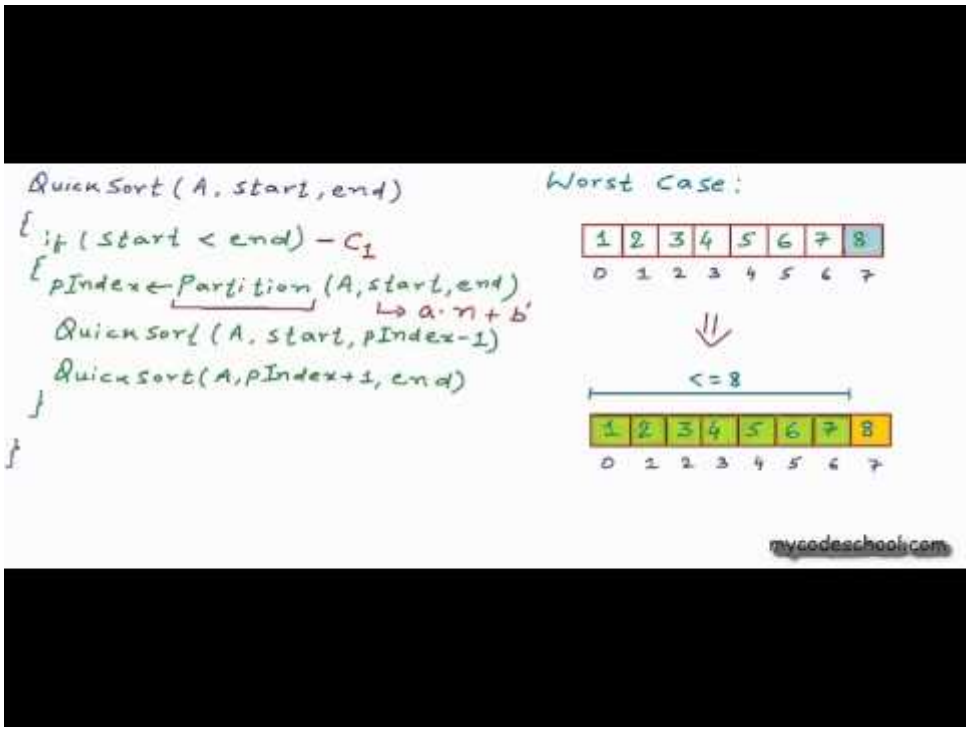
QuickSort

For tutorial and complexity analysis below videos are good

QuickSort Explanation



QuickSort Complexity Analysis



What is 3-Way QuickSort

In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for sub-arrays on left and right of pivot. Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}.If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences.

In 3 Way QuickSort, an array $arr[l \,..\, r]$ is divided in 3 parts:
a) $arr[l \,..\, i]$ elements less than pivot.
b) $arr[i + 1 \,..\, j - 1]$ elements equal to pivot.
c) $arr[j \,..\, r]$ elements greater than pivot.
See this for implementation.

How to implement QuickSort for Linked Lists?
QuickSort on Singly Linked List
QuickSort on Doubly Linked List

Can we implement QuickSort Iteratively?
Yes, please refer Iterative Quick Sort.

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.