Why Quick Sort preferred for Arrays and Merge Sort for Linked Lists?

- 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 has 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.

- 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 if we are given reference/pointer to the previous node. 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.

We can use Selection Sort as per below constraints:

- When the list is small. As the time complexity of selection sort is **$O(N^2)$** which makes it inefficient for a large list.
- When memory space is limited because it makes the minimum possible number of swaps during sorting.

We can use Bubble Sort as per below constraints:

- It works well with large datasets where the items are almost sorted because it takes only one iteration to detect whether the list is sorted or not. But if the list is unsorted to a large extend then this algorithm holds good for small datasets or lists.
- This algorithm is fastest on an extremely small or nearly sorted set of data.

Insertion Sort

This sorting algorithm is a simple sorting algorithm that works the way we sort playing cards in our hands. It places an unsorted element at its suitable place in each iteration.

We can use Insertion Sort as per below constraints:

- If the data is nearly sorted or when the list is small as it has a complexity of **O(N^2)** and if the list is sorted a minimum number of elements will slide over to insert the element at it's correct location.
- This algorithm is stable, and it has fast running case when the list is nearly sorted.
- The usage of memory is a constraint as it has space complexity of O(1).

We can use Merge Sort as per below constraints:

- Merge sort is used when the data structure doesn't support random access since it works with pure sequential access that is forward iterators, rather than random access iterators.
- It is widely used for external sorting, where random access can be very, very expensive compared to sequential access.
- It is used where it is known that the data is similar data.
- Merge sort is fast in the case of a linked list.
- It is used in the case of a linked list as in linked list for accessing any data at some index we need to traverse from the head to that index and merge sort accesses data sequentially and the need of random access is low.
- The main advantage of the merge sort is its stability, the elements compared equally retain their original order.

We can use Quick Sort as per below constraints:

- Quick sort is fastest, but it is not always **O(N*log N)**, as there are worst cases where it becomes **O(N^2)**.
- Quicksort is probably more effective for datasets that fit in memory. For larger data sets it proves to be inefficient so algorithms like merge sort are preferred in that case.

- Quick Sort in is an in-place sort (i.e. it doesn't require any extra storage) so it is appropriate to use it for arrays.

In place sorting algorthim					
	Time complexity			Space complexty	
Sorting Algorthim	best case	average ca	wores case	worse case	stable
Bubble sort	O(N^2)	O(N^2)	O(N^2)	O(1)	yes
Selection sort	O(N^2)	O(N^2)	O(N^2)	O(1)	no
Insertion sort	O(N)	O(N^2)	O(N^2)	O(1)	yes
Heap sort	O(NlogN)	O(NlogN)	O(NlogN)	O(1)	no
Quick sort	O(NlogN)	O(NlogN)	O(NlogN)	O(1)	?
not In place sorting algorthim					
	Time complexity			Space complexty	
Sorting Algorthim	best case	average ca	wores case	worse case	stable
Bucket sort	O(N+k)	O(N+k)	O(N^2)	O(N)	yes
Radix Sort	O(N+k)	O(N+k)	O(N+k)	O(N)	yes
PQ sort	O(NlogN)	O(NlogN)	O(NlogN)	O(N)	no
Merge sort	O(NlogN)	O(NlogN)	O(NlogN)	O(N)	yes/no it depen
radix Sort ParfectSequ	O(N)	O(N)	O(N)	O(N)	yes