Why is Quick Sort preferred for 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 is Merge Sort preferred 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 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.

Quick Sort is generally preferred for arrays because it has good cache locality and can be easily implemented in-place, which means it doesn’t require any extra memory space beyond the original array. In Quick Sort, we can use the middle element as the pivot and partition the array into two sub-arrays around the pivot. This can be done by swapping elements, and the pivot can be placed in its final position in the sorted array.

This process of partitioning is done recursively until the entire array is sorted. The cache locality of Quick Sort is beneficial because it minimizes the number of cache misses, which improves performance.

On the other hand, Merge Sort is generally preferred for linked lists because it doesn’t require random access to elements. In Merge Sort, we divide the linked list into two halves recursively until we have individual elements. Then, we merge the individual elements by comparing and linking them in a sorted order.

The advantage of Merge Sort for linked lists is that it doesn’t require random access to elements, which is not efficient for linked lists since we need to traverse the list linearly. Also, Merge Sort is a stable sort, which means it maintains the relative order of equal elements in the sorted list. This is important for linked lists, where the original order of equal elements may be significant. However, Merge Sort requires extra memory space for the merge step, which can be a disadvantage in some cases.