**General Assumptions:**

1. Input Size: The input array has n elements (n = 10,000 as default).
2. String Reference Size: In a 64-bit JVM, a String reference is 8 bytes.

**Merge Sort Memory Analysis:**

* Divides array into two halves recursively until subarrays have ≤1 element.
* Creates temporary arrays (left and right) during division, merged back into the original array.

**Temporary Subarrays:**

* Recursion depth is log₂(n). At each level, total temporary arrays hold n elements.
* Maximum temporary memory at any time: two arrays of n/2 elements each.
* Memory per element: 8 bytes (String reference).
* Total: (n/2 + n/2) \* 8 = n \* 8 bytes.
* For n = 10,000: 10,000 \* 8 = 80,000 bytes ≈ 80 KB.

**Recursion Stack:**

* Depth: log₂(n) ≈ 14 for n = 10,000.
* Stack frame: ~128 bytes per call.
* Total: 14 \* 128 = 1,792 bytes ≈ 1.79 KB.

**Total Additional Memory:**

* Total: 80,000 + 1,792 ≈ 81,792 bytes ≈ 81.8 KB.

**Quick Sort Memory Analysis:**

* Partitions array in-place around a pivot, recursively sorts subarrays.
* No additional arrays except temporary variables.

**Temporary Variables:**

* Partition method variables:
  + pivotValue: 8 bytes (String reference).
  + smallerElementIndex, currentIndex: 4 bytes each (integers).
  + temp for swapping: 8 bytes (String reference).
  + Method overhead: ~16 bytes.
  + Arguments (start, end): 4 bytes each.
* Total per partition call: 8 + 8 + 8 + 16 + 8 = 48 bytes.

**Recursion Stack:**

* Average case depth: log₂(n) ≈ 14 for n = 10,000.
* Each quickSort call:
  + Array reference: 8 bytes.
  + start, end: 4 bytes each.
  + Overhead: ~32 bytes.
* Total per call: 8 + 8 + 32 = 48 bytes.
* Partition call: 48 bytes.
* Total per level: 48 + 48 = 96 bytes.
* Total: 14 \* 96 = 1,344 bytes ≈ 1.34 KB.

**Total Additional Memory:**

* Total: ≈ 1,344 bytes ≈ 1.34 KB.

**Conclusion:**

For n = 10,000, Quick Sort uses less memory (~1.34 KB vs. ~81.8 KB) in the average case, making it more memory efficient.