```
public static void recursiveMerge(int[] a, int lo, int hi) {

	if (hi <= lo) return;
	int mid = lo + (hi - lo) / 2;
	recursiveMerge(a, lo, mid);
	recursiveMerge(a,mid+1,hi);


	// logic: keep track of the left and right endpoints and work towards the middle
	// i.e. imagine this as quicksort but with merging
	int x = lo;
	int y = mid+1;


	System.out.println(lo + " " + mid + " ");
	while (x <= mid && y <= hi)
	{
		if (a[x] < a[y] )
			x++;
		else
		{
			// shift the array in chunks, preserving final value
			int temp = a[x];
			int temp2 = 0;
			int temp3 = a[y];
			for(int i = x; i <= y; i++) {
				temp2 = a[i];
				a[i] = temp;
				temp = temp2;
			}
			a[x] = temp3;
			x=x+1;
			mid=mid+1;
			y=y+1;
		}
	}
```

1. In the best case scenario, we have an array in random order, We look at the yellow highlighted code:

$$Sort = Recursive_{Left} + Recursive_{Right} + Merge$$

The $Merge$ function, after completion of R_L and R_R will run $O(N)$ times, with the while loop updating bounding the updates from x to $N/2$ (0 to mid) and y increments to $N/2$ (mid to hi). So we have…

$$T(N) = T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + N = 2T\left(\frac{N}{2}\right) + N$$

This runs $NlogN$.

In the worst case, though, the recursive call will continually operate on a list of size $N-1$, ignoring either the left or right halves. This happens for an array in sorted order. So we end up continually doing:

$$Sort = Recursive_{N-1} + Merge$$

Which expands:

$$T(N) = T(N-1) + N$$

The solution to this is $O(N^2)$.

Does this use $O(N)$ extra space? No! There is no copy to an auxiliary array. We look at the red code to see that we end up shifting the array in chunks moving left to right if an element is out of order. The sort is stable because we shift elements in order and move them as a single block.