```
for(int i = 0; i < count; i++) {

	// runs N times
	for(int j = 0; j < a.length; j++) {

		int digit = (int) (a[j] % Math.pow(10, i+1)/Math.pow(10, i));
		digits[digit].Enqueue(a[j]);
		//System.out.println(digit);
	}

	int index = 0;

	//runs totally 10 times
	for(int x = 0; x < digits.length; x++) {

		// grabs pointer to memory 10 times
		ArbitraryQueue<Integer> current = digits[x];

		// only runs N times
		while(current.totalNodes > 0) {
			a[index] = current.Pop();
			index++;
		}

	}
```

1. The runtime of this algorithm is bounded by the sequence of loops above. The complexity will end up being $Loop1 * (Loop2 + Loop3)$:
   a. $Loop1$ runs in time proportional to $count$. $Count$ represents the number of digits in the largest number – or $\log_2 \max(a) + 1$.
   b. $Loop2$ runs in time proportional to $a.length$ -- or the size of the input array, N.
   c. $Loop3$ runs in time proportional to the number of queues used to store the array values. This is equal to the total number of digits possible in the base representation. Since there are 10 digits in base 10, this will run 10 times.

   In total, this will run:

$$Loop1 * (Loop2 + Loop3) = \log_2(\max(a))\,(N + 10)$$

   In Big-Oh notation, it would be:

$$O(Nlog_2(\max(a))$$

2. This algorithm already used $O(N)$ extra space at this point:

```
	for(int j = 0; j < a.length; j++) {
		int digit = (int) (a[j] % Math.pow(10, i+1)/Math.pow(10, i));
		digits[digit].Enqueue(a[j]);
		//System.out.println(digit);
}
```

For every value in input array a, we push it onto a queue corresponding to a digit in base-10.

3. The switch to constant extra space requires eliminating the use of queues to store every instance of each digit of base 10 from the original array. This means we need to swap in-place without comparing to values. The logic for this is:
   a. Create an array of size 10, one for each digit 0-9, that is used to count every instance of that digit we find when sorting by ones, tens, hundreds, etc. place.

b. Create an array of size 10, one for each digit 0-9, that is used to calculate the position we should be placing a given digit at in the array. It is initialized by positions[i] = counts[i-1] + positions[i-1], where positions[0] = 0. (Digits of position 0 should go at the start of the array)

So the updated code looks like:

```
for(int i = 0; i < count; i++) {

	// this runs at most 10 times
	for(int x = 0; x < counts.length; x++) {
		counts[x] = 0;
		positions[x] = 0;
	}


	// runs N times
	for(int j = 0; j < a.length; j++) {

		int digit = (int) (a[j] % Math.pow(10, i+1)/Math.pow(10, i));
		counts[digit] = counts[digit]+1;
		//System.out.println(digit);
	}
	// runs 9 times
	for(int x = 1; x < counts.length; x++) {
		positions[x] = positions[x-1] + counts[x-1];
	}

	// runs N times
	for(int j = 0; j < a.length; j++) {

		int digit = (int) (a[j] % Math.pow(10, i+1)/Math.pow(10, i));
		int position = positions[digit];
		int temp = a[j];
		a[j] = a[position];
		a[position] = temp;
		if(positions[digit]+1 <  a.length)
		positions[digit] = positions[digit]+1;

	}
```

The complexity becomes: $Loop1 * (Loop2 + Loop3 + Loop4)$.

a. $Loop1$ runs the number of digits of the largest number in the array. This is $\log_2 \max(a) + 1$.
b. $Loop2$ runs a.length, or N, the length of the input.
c. $Loop3$ runs counts.length, which is the array that is of constant size 10 – the number of digits in base 10 (0-9).
d. $Loop4$ runs N again, a.length.

In total, the complexity is:

$$[\log_2 \max(a) + 1](N + 10 + N)$$

Which is about:

$$\sim 2N log_2 \max(a)$$

The constant extra space is achieved by using 2 arrays of size 10 each and performing in-place swaps corresponding to a series of updated position counters.

Stability

The first sort is stable. The first sort enqueues digits of equal position onto the queue and moves them as an in-order chunk. They are dequeued, together, off of the same queue. The second one is stable since the positional counter updates with every swap. If you find elements of equal order they are placed one to the right of each other until you hit the end of the array.