Code with non-constant space usage

```
void Problem1(int a[], int arraySize) {
       //int rem[5];
       int data[10]; //constant time
       int cntzero = 0; //constant time
       int cnt2 = 0; //constant time
       int val; //constant time
       int tmp3; //constant time
       int base = 0; //constant time
       int divider; //constant time
       int valcnt = 0; //constant time
       int output[arraySize]; //constant time
       int i, j, k, l, m, n = 0;
       //cout << " This ran" << endl;
       while (cntzero < arraySize) {// runs k+1 times
              cnt2 = 0;//constant time
              //cout << " This ran" << endl;
              for (i = 0; i < 10; ++i) {//resetting data to zero//runs b times (in this
problem b=10)
                     data[i] = 0;

              }
              divider = pow(10, base); //constant time
              cntzero = 0; //constant time
              for (j = 0; j < arraySize; ++j) {//runs n times – one of the biggest
bottlenecks
                     //     rem[j] = (a[j] / divider) % 10;
                     ++data[(a[j] / divider) % 10];
                     if ((a[j] / divider) % 10 == 0) {
                            ++cntzero;
                     }

              }
              ++base;

              if (cntzero < arraySize) {
                     for (i = 1; i <= 9; ++i) {//runs b-1 times
                            data[i] += data[i - 1];
                     }
                     // Build the output character array


                     for (i = arraySize-1; i >= 0; --i) //runs n times – one of the
biggest bottlenecks

                     {//this showcases where it remains stable
                            output[data[((a[i] / divider) % 10)] - 1] = a[i];

                            data[(a[i] / divider) % 10]--;
                     }

                     // Copy the output array to arr, so that arr now
                     // contains sorted characters
                     for (i = 0; i < arraySize; ++i) {{//runs n times – one of the
biggest bottlenecks

                            a[i] = output[i];
```

```
                        }
                }

        }
}//Total Run time = Constant code before 1st loop + K (3n + constant in while loop) =
O(kn)
```

The run time of this algorithm when the sort does not use constant space is equal to O(kn) where k is equal to the number of digits the largest integer contains plus one. For example, if the biggest integer was 5321, k would be equal to 5. The code snippet shown above shows a breakdown of the run time. The best-case situation is when the integers are all low digit numbers (1-3). In this case the sort runtime is O(n). The worst case is when the number of digits that the largest integer is the maximum possible value. In this situation the k would be equal to log(k), which means the worst-case time complexity would be O(n*log(n)). If the base of each integer was not guaranteed to be 10 (e.g. all the numbers are base 6), the general time complexity would become O(k(n+b)) where is equal to the base of the integers. In the worst-case situation, the time complexity would become O((n+b)*log(n))

The sort shown in this code is stable. This is because the code always takes elements from the original array and puts them in the output array in the order they were originally placed (when there are duplicates). For example, if you were sorting (37, 7, 23, 4) by the last digit of each integer, the array after the sort completes would be (23, 4, 7, 37).

Code with constant space usage

```
void Problem1(int a[], int arraySize) {
        int data[10];
        int cntzero = 0;
        int cnt2 = 0;
        int val;
        int tmp3;
        int base = 0;
        int divider;
        int valcnt = 0;
        int i, j, k, l, m, n = 0;
        //cout << " This ran" << endl;
        while (cntzero < arraySize) {
                cnt2 = 0;
                //cout << " This ran" << endl;
                for (i = 0; i < 10; ++i) {//resetting data to zero
                        data[i] = 0;

                }
                divider = pow(10, base);
                cntzero = 0;
                for (j = 0; j < arraySize; ++j) {
                        //      rem[j] = (a[j] / divider) % 10;
                        ++data[(a[j] / divider) % 10];
                        if ((a[j] / divider) % 10 == 0) {
                                ++cntzero;
                        }

                }
                ++base;
```

```
		//cout << " This ran" << endl;
		for (k = 0; k < arraySize; ++k) {//printing for loop
			cout << a[k] << ", ";

		}
		cout << endl;

		if (cntzero < 5) {


			for (l = 0; l < 10; ++l) {

				while (data[l] != 0) {
					if (data[l] != 0) {
						val = l;
						valcnt = 0;

						for (m = 0; m < arraySize; ++m) {
							if ((a[m] / divider) % 10 == val) {
								//cnt2 = cnt2 - valcnt;

								tmp3 = a[cnt2];//in place addition
								a[cnt2] = a[m];//in place addition

								a[m] = tmp3;//in place addition
								++cnt2;
								++valcnt;

							}
						}
					}// the code no longer uses an additional array to
temporary sort the values or an array that uses. The code now swaps the values in the
same array.
					data[l]=0;

				}
			}
		}

	}
}
```

The run time of the algorithm using constant space (shown above) is the same as the run time of the algorithm using O(n) space. This version of the code checks to see if the value of the current digit being sorted in the array is equal to the current index value of data (if the contents of that index are not equal to 0). If they are equal, that value is swapped with the value of the first element in the array (In this situation cnt2=0). After that the value of cnt2 increments by 1 and the index of data also increments by 1. This process repeats until the array is sorted by that digit. This implementation is the superior one because it does not increase the time complexity, but it reduces the space complexity. This version of the algorithm is also stable for a similar reason as the one mentioned in the previous section.