Quiz No. 07:

# Quiz 07 Extra Credit

submitted to:

Professor Joseph Picone
ECE 3822: Software Tools for Engineers
Temple University
College of Engineering
1947 North 12th Street
Philadelphia, Pennsylvania 19122

April 6, 2018

prepared by:

Ryan Anstotz
Email: tuc45516@temple.edu

Emily Lofing
Email: emilylofing@temple.edu

## A. DESCRIPTION

For this assignment, the objective was to test the efficiency of six different algorithms that output equivalent data. The function was a "count duplicates" implementation that input an array of numbers, and returned the number of duplicates in the input array. The six different algorithms were confirmed to output accurate data. However, each algorithm's run time was noticeably different when data sets were scaled. This paper investigates the runtime and memory usage of each algorithm with respect to time and computer memory.

The algorithms were tested using Python. Python is an interpreted, object-oriented programming language that uses dynamic typing. Python is especially useful for rapid-prototyping. While Python has its advantages, since it is an interpreted language, it executes computations much slower than a compiled language such as C. The objective was to compare algorithms relatively, therefore, Python was an appropriate and effective tool.

To compare the algorithms for run time and memory usage, the plan was to develop two Python scripts. So, each set of the six algorithms was run through each script (12 scripts total). Let me describe the design of each script.

The first script was designed to time one of the six algorithms as specified through a command line argument. Then the script would be executed five more times, one per each algorithm, and the results sent to an output file. The second script would do the same for memory. For the algorithm specified, the script iterated over the execution of the algorithm 10 times, each time storing the time or memory required to execute. Each of these 10 runs was iterated over by one of five randomly generated lists. The lists were of length 100, 1000, 10,000, 100,000, and 1,000,000 integers and served as the "count duplicates" function argument. The random integers within these random lists were comprised of a range of ¼ the sample size. So, for the list of 100 integers, there may be 25 different integers within that list. This was done to ensure an adequate sample size. After the 10 runs were iterated over a specific sample size, the average of the 10 runs was stored outside of the random list loop. At this point we have 10 averaged runs in a list of five elements corresponding the random integer lists input. Then, this experiment was executed within a for loop of 10 additional runs, to confirm a smoothed average value for the time and memory.

The design of the scripts are as follows. First, a dictionary was implemented to accept function calls from the command line. This design created a string as the key and the function as the value. This allowed for a simple command line interface implementation. Next, the experiment data, such as the lengths of the for loops, and lengths of the random data size, were stored into variables so that the program could be thoroughly tested for accuracy.

Time was recorded with a time function from the "default_timer" library placed before and after the function call. This method provided consistent results after testing. For memory, we placed a function to pull the process ID called "os.getpid()" and stored it to a variable local to the function. Then, from the library "from memory_profiler import memory_usage", we used a function called "memory_usage()" and placed the PID variable as the argument. Also, we removed this memory implementation from the function and placed it directly after the function call in the "main" function, and the memory usage produced similar results. This implementation seemed to scale well on the 1,000,000 size data set, however appeared to produce lesser differences in the smaller data sets. This phenomenon is explored in more depth in the results section.

Below, in Figure 1 we have provided the algorithms that were tested.

**Solution 1:**
```
def countDuplicates(numbers):
    d = {}
    for val in numbers:
        d[val] = d.get(val, 0) + 1
    return sum(d[i] > 1 for i in d)
```
===
**Solution 2:**
```
def countDuplicates(numbers):
    dupVals = []
    for i in range(0, len(numbers)):
        for j in range(i+1, len(numbers)):
            if numbers[j] == numbers[i] and numbers[j] not in dupVals:
                dupVals.append(numbers[j])
    return len(dupVals)
```
===
**Solution 3:**
```
def countDuplicates(numbers):
    temp = []
    foo = 0
    numberz = set(numbers)
    for num in numberz:
        temp.append(numbers.count(num))
    for num in temp:
        if num > 1:
            foo += 1
    return foo
```
===
**Solution 4:**
```
 def countDuplicates(numbers):
  x=[]
  for n in set(numbers):
   count = numbers.count(n)
   if count > 1:
    x.append(n)
 return (len(x))
```
===
**Solution 5:**
```
def countDuplicates(numbers):
    counter = 0
    if len(numbers) < 2:
        return counter
    else:
        numbers.sort()
        dup = 0
        for i in range(1,len(numbers)):
            if ((numbers[i-1] == numbers[i]) & (dup == 0)):
                counter = counter + 1
                dup = 1
            elif numbers[i-1] != numbers[i]:
                dup = 0
    return counter
```
===
**Solution 6:**
```
def countDuplicates(numbers):
    allDupes = [x for x in numbers if numbers.count(x) >= 2]
    uniqueDupes = list(set(allDupes))
    numberOfDupes = len(uniqueDupes)
    return numberOfDupes
```

**Figure 1– Tested Algorithms (six)**

## B. RESULTS

The original plan was to execute all six algorithms against the full random list data set for both time and memory. However, the programs ran for over 24+ hours and only two algorithms executed to completion for both time and memory. Therefore, the dataset was reduced to 10, 100, 1000, 10,000. Also, the random sample size was changed from ¼ to ½ in an effort to increase run time and ensure that all data was proportional (i. e. 10 is not divisible by 4, whereas the remainder of the data set is).

Since only Algorithm 1 and 5 executed to completion, it can be stated that these algorithms are the most efficient. Below, in Figure 2 is the plot of the full sample size of Algorithm 1 and 5. For this figure and the following, the axes will be log-log. After Figure 1, the data may be gleaned in Table 1.

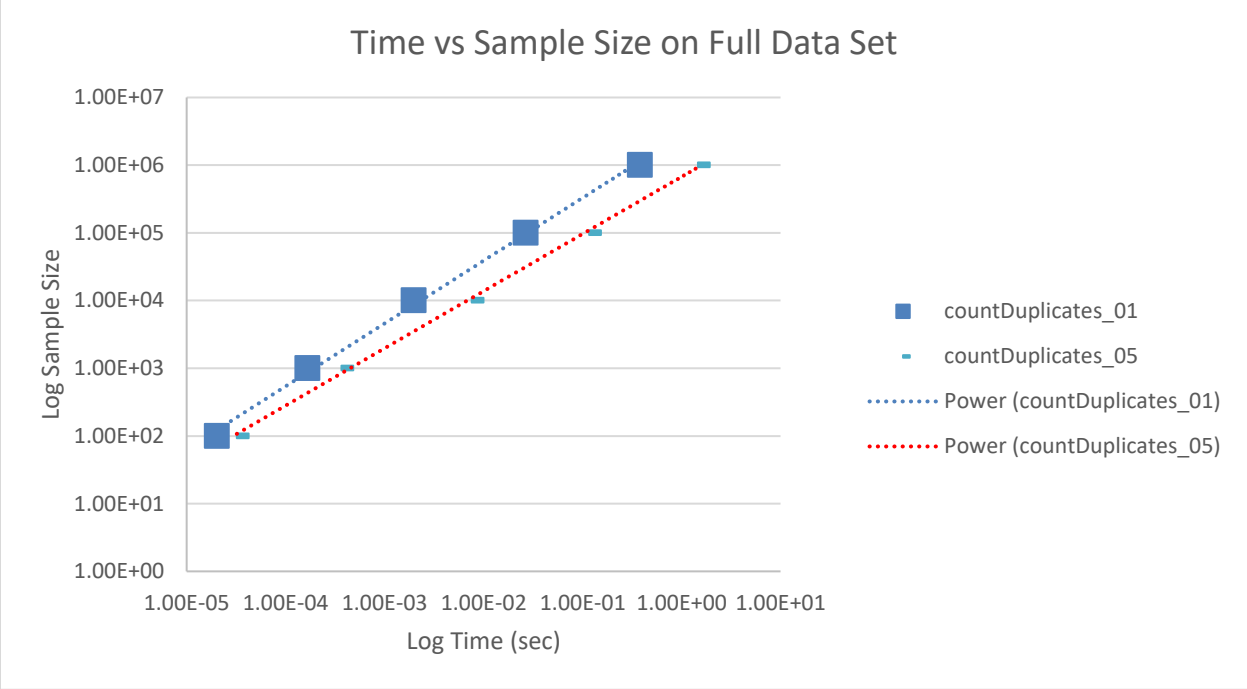


**Figure 2 – Time vs Sample Size on Full Data Set**

**Table 1 – Raw Time Data on Full Data Set**

| Sample Size -> | 1.00E+02 | 1.00E+03 | 1.00E+04 | 1.00E+05 | 1.00E+06 |
|---|---|---|---|---|---|
| Algorithm | | | | | |
| countDuplicates_01 | 2.02E-05 | 1.67E-04 | 1.97E-03 | 2.66E-02 | 3.81E-01 |
| countDuplicates_02 | | | | | |
| countDuplicates_03 | | | | | |
| countDuplicates_04 | | | | | |
| countDuplicates_05 | 3.21E-05 | 3.67E-04 | 7.56E-03 | 1.17E-01 | 1.46E+00 |
| countDuplicates_06 | | | | | |

By this time metric alone, Algorithm 1 performs the best out of all the algorithms, with Algorithm 5 coming second. Below, the smaller data set is timed and graphed; first depicted in Figure 3, then the data in Table

2. As a note, time is recorded in seconds.

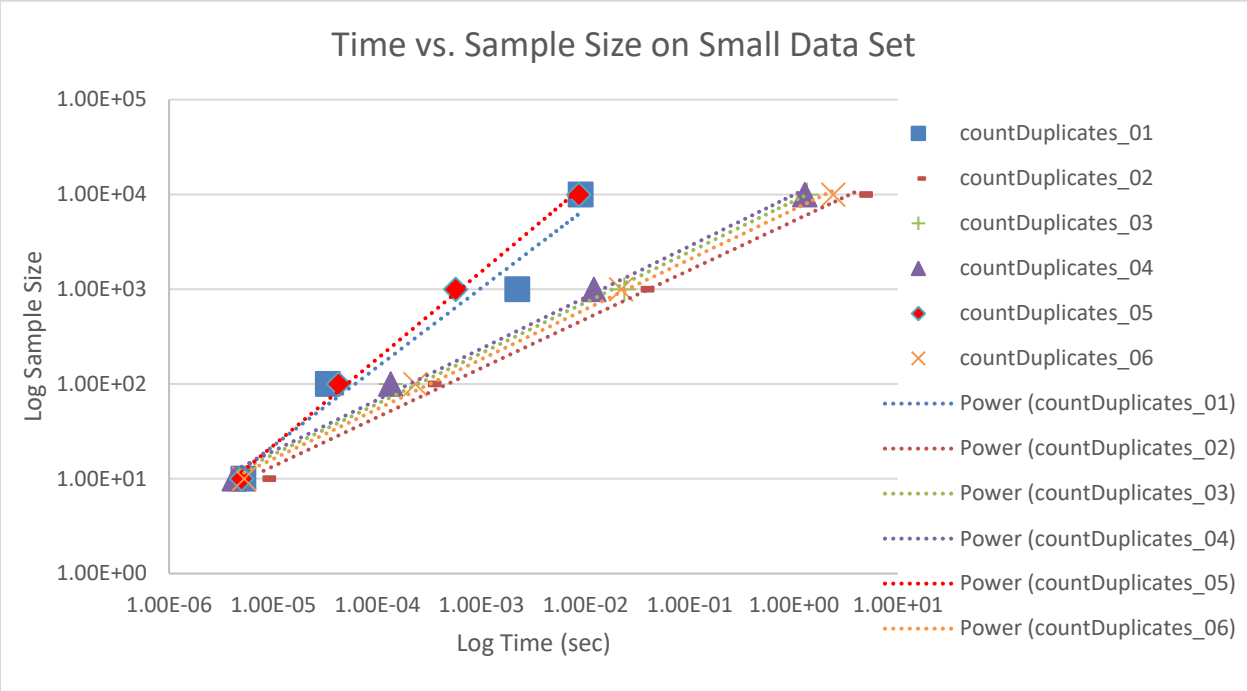


**Figure 3 – Time vs Sample Size on Small Data Set**

**Table 2 – Raw Time Data on Small Data Set**

| Sample Size -> | 1.00E+01 | 1.00E+02 | 1.00E+03 | 1.00E+04 |
|---|---|---|---|---|
| Algorithm | | | | |
| countDuplicates_01 | 5.17E-06 | 3.35E-05 | 2.21E-03 | 9.03E-03 |
| countDuplicates_02 | 8.06E-06 | 3.12E-04 | 3.44E-02 | 4.35E+00 |
| countDuplicates_03 | 4.79E-06 | 1.34E-04 | 2.36E-02 | 1.34E+00 |
| countDuplicates_04 | 4.22E-06 | 1.34E-04 | 1.21E-02 | 1.28E+00 |
| countDuplicates_05 | 4.96E-06 | 4.23E-05 | 5.66E-04 | 8.64E-03 |
| countDuplicates_06 | 5.27E-06 | 2.32E-04 | 2.19E-02 | 2.40E+00 |

For the smaller sample size, Algorithm 5 if faster than Algorithm 1 by a tight margin at the 10,000 sample size. The interpretation of this data indicates that the algorithms from fastest to slowest are: Algorithm 5, 1, 4, 3, 6, then 2.

The memory incurred by each algorithm was much more difficult to interpret due to the close clustering of the data points extracted. Algorithm 1 and 5 were the only memory tests to reach completion and below in Figure 4 and Table 3 are the results. Memory was recorded in MiB.

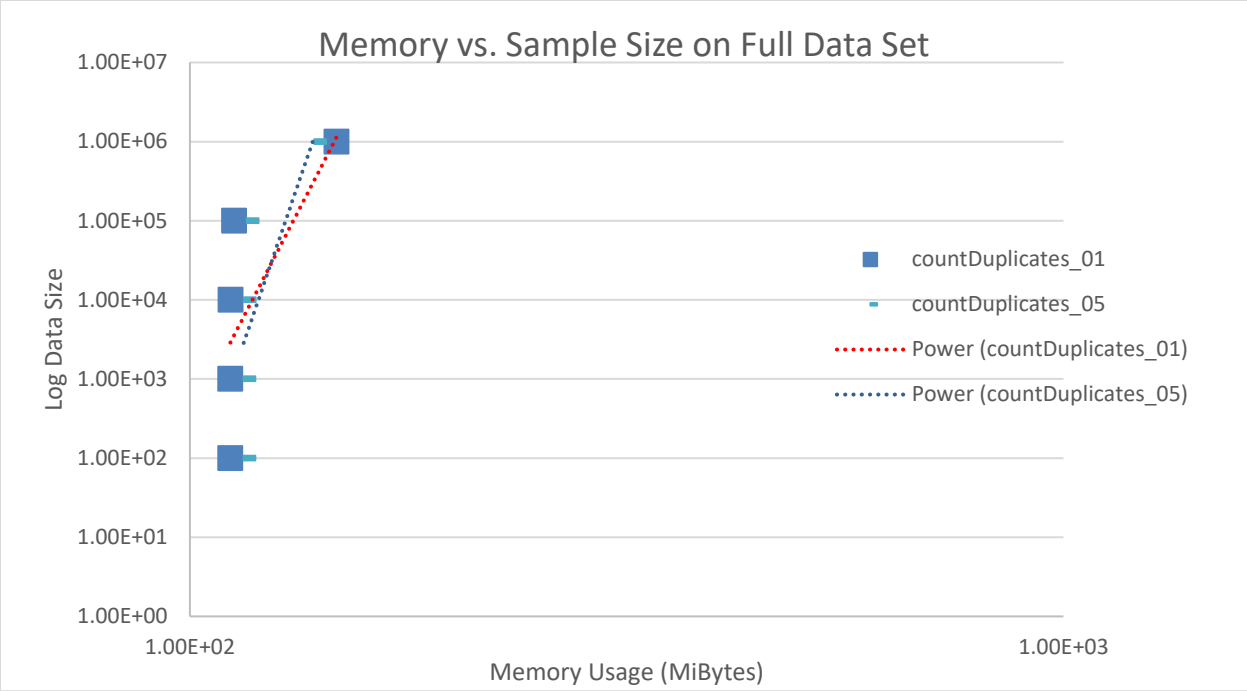


**Figure 4 – Memory vs Sample Size on Full Data Set**

**Table 3 – Raw Memory Data on Full Data Set**

| Sample Size -> | 1.00E+02 | 1.00E+03 | 1.00E+04 | 1.00E+05 | 1.00E+06 |
|---|---|---|---|---|---|
| Algorithm | | | | | |
| countDuplicates_01 | 1.11E+02 | 1.11E+02 | 1.11E+02 | 1.12E+02 | 1.47E+02 |
| countDuplicates_02 | | | | | |
| countDuplicates_03 | | | | | |
| countDuplicates_04 | | | | | |
| countDuplicates_05 | 1.15E+02 | 1.15E+02 | 1.15E+02 | 1.16E+02 | 1.39E+02 |
| countDuplicates_06 | | | | | |

As shown in the data, the memory footprint of these functions stays roughly the same over the 100, 1000, and 10,000 sample sizes. However, it increases exponentially between 10,000 and 1,000,000. Algorithm 1 had the smallest memory footprint according to this analysis.

Next, the smaller data set was tested. Since this was only tested up to the 10,000 data set, the data was very tightly clustered. Below is Figure 5 and Table 4 containing the data.

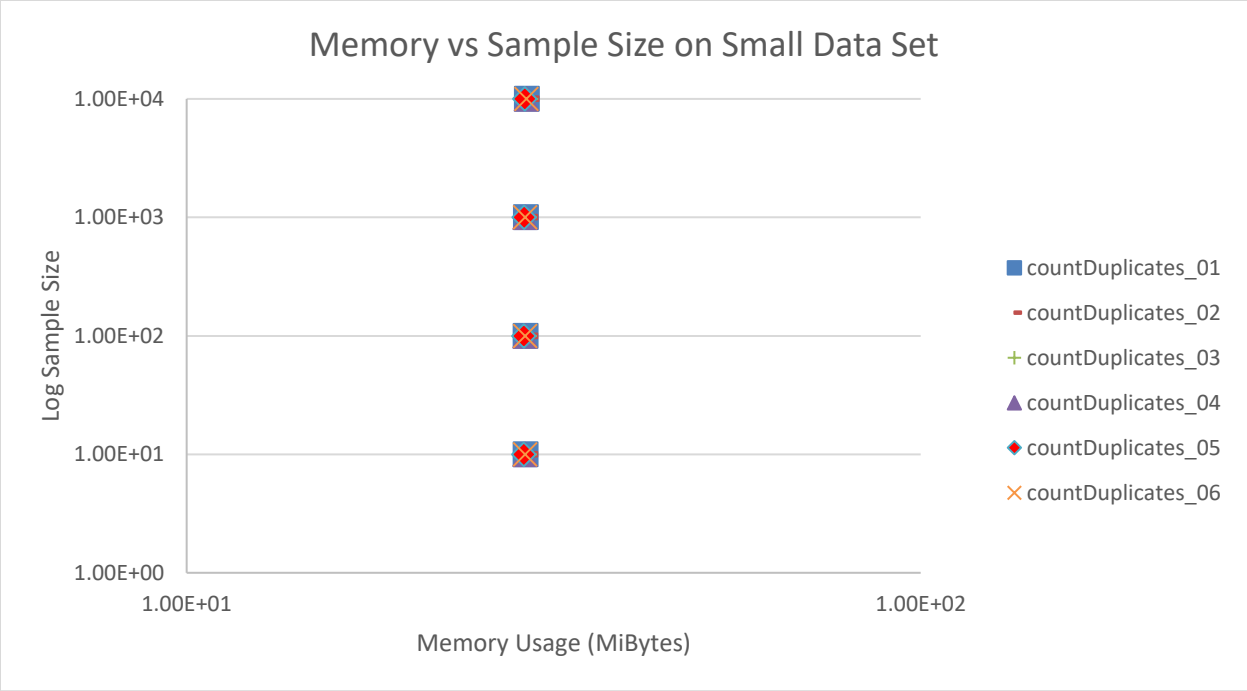


**Figure 5 – Memory vs Sample Size on Small Data Set**

**Table 4 – Raw Memory Data on Small Data Set**

| Sample Size -> | 1.00E+01 | 1.00E+02 | 1.00E+03 | 1.00E+04 |
|---|---|---|---|---|
| Algorithm | | | | |
| countDuplicates_01 | 2.90E+01 | 2.90E+01 | 2.90E+01 | 2.91E+01 |
| countDuplicates_02 | 2.90E+01 | 2.90E+01 | 2.90E+01 | 2.91E+01 |
| countDuplicates_03 | 2.89E+01 | 2.89E+01 | 2.89E+01 | 2.90E+01 |
| countDuplicates_04 | 2.89E+01 | 2.89E+01 | 2.89E+01 | 2.90E+01 |
| countDuplicates_05 | 2.88E+01 | 2.88E+01 | 2.88E+01 | 2.89E+01 |
| countDuplicates_06 | 2.89E+01 | 2.89E+01 | 2.89E+01 | 2.90E+01 |

As shown, the data is nearly identical. Also, the memory footprint of Algorithms 1 and 5 changed from the full data set. This may be a result of other processes running. While the data is tightly clustered, it does trend upward with the increase in data size.

## C. SUMMARY

This exercise demonstrated results and techniques for measuring algorithm efficiency. Each algorithm implemented varying data structures to accomplish the task of finding duplicates in a Python list of integers. Algorithm 1 implemented a dictionary (O(1) for a dictionary). This dictionary is efficient in time for the task, but with that efficiency should come an increase in memory. Algorithm 2 is the slowest in terms of time (uses lists that are traversed at O(n) per each list) but may be more memory efficient since lists are implemented rather than dictionaries. The Time and memory should be inversely proportional. Our memory data did not clearly accentuate this concept since the small data size had data points so tightly clustered. An important next-step would be to implement a separate memory profiler in an effort to gather additional data.

However, our time data was clear and provided a greater understanding of the time efficiencies of the functions at hand.