1c.

So my program, including the Porblem1Sort function, varies in time elapsed depending on the number of elements in my array and the number of digits contained in each element. The cost of operations is defined as how much time did each operation cost the compilation. Some of the operations found in my program are: integer comparisons, variable declarations, array element access, and assignment statements. Being that my code uses 5 bubbleSort operations, the bubble sort takes up the most time and memory in order to execute and uses O(N^2) runtime.

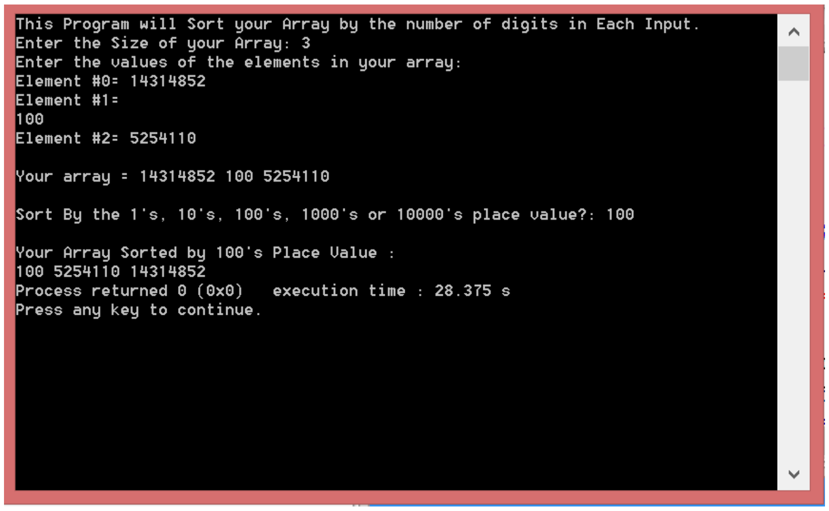
- Integer comparisons in my program:
    - Integer (int i, int j) < sizeEntered (shows up 2 times)
    - Integer (int k, int f) < arraySize (shows up 11 times but we are only considering 3)
        - For this, I am only considering the 2 times it shows up in my function because my program has 5 conditional statements where this comparison shows up, but only 1 conditional statement is fulfilled during compilation. Each conditional statement compares 2 times.
        - The 3rd time it shows up is when my program outputs the final array
    - a[f]%10 > a[f+1]%10
    - a[f]%100 > a[f+1]%100
    - a[f]%1000 > a[f+1]%1000
    - a[f]%1000 > a[f+1]%1000
    - a[f]%1000 > a[f+1]%1000
        - only consider one of these because only one of these are executed during a compilation
- Variable declarations in my program:
    - int sizeEntered
    - int values
    - int entered[sizeEntered] = {0}
    - int hold
    - int decimal
- Array element access' in my Program:
    - entered[i] (shows up two times)
    - a[q] shows up once
    - a[f] shows up six times (six times in each conditional statement)
- Assignment statements in my program:
    - int g = 0
    - entered[i] = values
    - hold = a[f] (5 times, once in each conditional statement)
    - a[f] = a[f+1] (5 times, once in each conditional statement)
    - a[f+1] = hold (5 times, once in each conditional statement)

1d.

A stable sort is a sort where if my sorting is looking at each element one by one, and they're comparing two elements with the same value in the same place value being observed by the algorithm, and the algorithm sorts them in the same order they were inputted, instead of looking at the next variable and determining where they should be in the sort due to the second variable in each of the elements. My code is stable because it obeys the definition and does indeed sort two integers with the same value in the place value being observed in the order it was inputted. Place value in this context is the 10th, 100th, etc. value of two numbers. *I will attach an example output with this document*

Where the code remains stable in the code snippet:

```
for (int k = 0; k < arraySize - 1; k++)
    for (int f = 0; f < arraySize - k - 1; f++)
        if (a[f]%10 > a[f+1]%10)
        {
        hold = a[f];
        a[f] = a[f+1];
        a[f+1] = hold;
        }
```



```
This Program will Sort your Array by the number of digits in Each Input.
Enter the Size of your Array: 3
Enter the values of the elements in your array:
Element #0= 14314852
Element #1=
100
Element #2= 5254110

Your array = 14314852 100 5254110

Sort By the 1's, 10's, 100's, 1000's or 10000's place value?: 100

Your Array Sorted by 100's Place Value :
100 5254110 14314852
Process returned 0 (0x0)   execution time : 28.375 s
Press any key to continue.
```

1e.

If your algorithm did not use constant extra space, what major change to the algorithm does using O(1) extra space provide?

My algorithm does not use extra space being that every operation happens in place, and no external storage place is used, so algorithm O(1) doesn’t change too much of my algorithm since it does not need the extra space in the first place