This is the Question / Answer section of a Siemens Assignment.

Please refer to the Questions_Siemens.py file for the code.

Answers are below in PURPLE.

Question 1 (code required):

You have two arrays of integers.

```
a = [47,24,95,184,13,3,12,18]
b = [83,9,32,29,52,90,108,14]
```

Provide 2 or 3 different ways to calculate the pair of values (one value in each array) with the smallest non-negative difference.

Return the elements in the array and the smallest non-negative difference.

For example, the answer with array `a` & `b` above is `b = 14` and `a = 13` because their difference is `14-13 = 1`, which is the smallest non-negative difference.

We are not looking for the most optimal code, but a variety of methods you could use calculate the smallest different and why you would select one method over another.

There is some ambiguity in the definition here about "non-negative difference". The uncertainty comes because it is not stipulated which array is subtracted from the other array. Is "a" subtracted from "b" or is "b" subtracted from "a" ? Due to this ambiguity, I have chosen to just treat the "non-negative difference" as the absolute difference.

Please refer to the **Questions_Siemens.py** file from https://github.com/mdsmit5x/Siemens_Jan_7_2022/

Lines 65, 66, 119, and 120 show the other interpretation of it always being "a – b"

I have selected three methods for the solution. A fourth method ("A better approach") is mentioned in the answer to Question 3 below. Refer to the Question_1 Class for these methods:

```
find_chance_of_zero_for_answer()

This function returns the probability of two arrays having the same number based on the
size of the arrays and the range of the random numbers in the arrays.  This function is
useful to guess whether to bother running more time-intensive search functions.  If there
is a very high probability of the two arrays having at least one element matching, then
the user may choose to just assume that the answer is zero rather than taking time to get
the actual answer.

runStupidBruteForce()

This function does a search across every element in one array against every element of
the other array.

runSortedDifference()

This function takes advantage of the behavior of a search for differences between two
arrays' elements.  By sorting the arrays first, the elements are compared between the two
arrays by taking advantage that the arrays are sorted.
```

Question 2 (code required):

Create 2 lists of random integers between 1 and 1,000,000. Each list should be 5,000 integers long.

Apply the multiple methods you derived in Question 1 with the 2 new arrays of integers you calculate in this question.

Which algorithm performed best? Which algorithm performed worst? And why? What is the Big0 notation for each of your methods?

```
Please refer to the Questions_Siemens.py file from
https://github.com/mdsmit5x/Siemens_Jan_7_2022/

find_chance_of_zero_for_answer()

This function indicates that the chance of two random arrays (as indicated above) is very
high that there will be at least one number that matches.  The chance is calculated at
99.9996%

runStupidBruteForce()

This function returned zero within an average 1.6 seconds after 100 trials

At this small upper bound (relative to the array size),  there is a problem in looking at
this result, because of the high probability of finding zero differences.  The timing
will not be consistent or predictable due to the random chance of where the numbers might
occur, since the function returns immediately if two matching elements are found.
```

runSortedDifference()

This function returned zero within an average 0.0019 seconds after 1000 trials. Nearly the entire time for this search was from the initial sorting of the two arrays.

At this small upper bound (relative to the array size), there is a problem in looking at this result, because of the high probability of finding zero differences. The timing will not be consistent or predictable due to the random chance of where the numbers might occur, since the function returns immediately if two matching elements are found.

In order to realistically see the timing of these functions, I ran a test again with an upper bounds of 1,000,000,000,000

For these new parameters, I got the following results for arrays with 5,000 random elements:

find_chance_of_zero_for_answer()

This function indicates that the chance of two random arrays as indicated above is very low that there will be at least one number that matches. The chance is calculated at 0.125%

runStupidBruteForce()

This function returned within an average of 35 seconds.

runSortedDifference()

This function returned within an average of 0.035 seconds.

In BigO notation:

find_chance_of_zero_for_answer() is of order N

runStupidBruteForce() is of order N^2

runSortedDifference() is of order N

It might be questioned why we would even bother with find_chance_of_zero_for_answer() since runSortedDifference() is also of order N.

The answer is: find_chance_of_zero_for_answer() is ten times faster than runSortedDifference()

Question 3 (no code required. We're only looking for a written response):

Conceptually, what are the different tools such as code libraries or infrastructure that could help you find the smallest non-negative difference between 1 million **lists** that are 5,000 integers long?

In this talk, I will assume that "lists" means "arrays", since that is the context used previously in this document. I am also assuming that the goal is to find the smallest non-negative difference between all elements of any two arrays. If we were to find any two arrays that had any element that was in both arrays, then the difference would be zero and we could stop searching further arrays.

The size of the memory needs to be taken into account. 1 million lists with 5,000 integers is 5 billion objects. Each int object in Python takes 24 bytes. This would mean taking a total space of more than 120 billion bytes if we loaded all the lists into memory at the same time in Python. Although I have done this project in Python, I would recommend moving to the C language for efficiency in dealing with 5 billion objects

If we compare all 1 million lists with each other, then the total comparisons would be

1,000,000 ! = 2.8 x 10 ^ 456573

Unless the upper bound is very huge, the probability of finding a difference of zero between all these compares is practically speaking 100%, so we can stop when we find the zero difference the first time. If we were to use the upper bound of 1 million (as specified in Question 2), only a few comparisons are needed. With near certainty (99.9996%), it will not take many comparisons to find two lists that have some elements that are equal.

If we assume a very huge upper bound such that zero differences between lists are unlikely, then the strategy is very different.

I would recommend saving the lists onto a storage device such as a hard drive on in the cloud. If we are to do comparisons, then it is important to take clumps of arrays into memory to do comparisons. Since each comparison will take an average of 3 milliseconds,

A better approach for many arrays:

Instead of doing this amazingly large amount of comparisons, let us instead not compare individual arrays against each other. For instance, if we were to generate a master list of all known numbers, then we could know when a number appeared more than once from a different array and then stop the comparisons. We could also find the two numbers which are closest together but in different arrays.

Each lists' elements would be inserted into a sorted master list. The element must also be flagged with its array number. In this case we could use a binary search to quickly find where the new element belongs and whether there is already an array with that element. If we find a match with a different array flag, then we can stop, since we have found the zero difference.

If after adding the 5 billion elements there is still no zero difference, then we would just traverse the ordered list of 5 billion elements and find the smallest difference between adjacent elements which have different array flags.

If we are using the "C" language, then I would use the <stdlib.h> library in order to use bsearch() as well as the <stdio.h> library for file reading and writing.

We will have problems with the size of the sorted array, because bsearch() will probably not handle that many elements. Documentation indicates that it is limited to 2 billion elements. Instead, we will have to split the master list into separate parts of adequate size. I would recommend creating a class/struct that will encapsulate the full functionality of inserting, searching, and traversing this master array. Internally it would use file systems and binary searches across multiple files to present externally a single cohesive array.

If this function will be needed frequently, then it would make sense to do this in a multi-threaded environment due to the large quantity of processing. To facilitate multi-threading, I would create the class so that the internal array is split into 4,000 separate memory spaces. This might be 4,000 files or tables or list objects. Those memory spaces would have locked semaphores to control the read/write access to them. By splitting the memory space 4,000 times, the memory block sizes would be 15Mbytes, which is a manageable size.

After setting up the class, externally it will still be viewed as one cohesive master list of all numbers seen in the 1,000,000 lists. The number of threads used to input into the master list will depend on resources including time. For instance, 100 threads would on average be blocked 10% of the time, but would be 90 times faster than a single-threaded system.