**Question 2: Which algorithm performed best? Which algorithm performed worst? And why? What is the Big0 notation for each of your methods?**

I have used two different ways to solve question # 1. The 1st method has a complexity of O(n^2) because of the nested FOR loops. The outer loop takes O(n) time and the inner loop takes O(n) time, so their composition is O(n) \* O(n) = O(n^2).

The 2nd method only uses 1 while loop and a pointer to increment position in both sorted arrays. The complexity of the while loop is O(n) while that of sort() function is O(nlogn), total algorithm complexity is O(n) + O(nlogn) = O(nlog n)).

I’ve also added a timer to calculate the time required for execution using the library *timeit.* Undoubtedly, the 2nd algorithm performed the best while the 1st algorithm performed the worst. On my computer, the 2nd algorithm takes 0.001 sec while the 1st algorithm takes 0.4 sec for execution (this will vary between computers).

The 1st algorithm had 2 nested For-loops which makes it computationally expensive and inefficient, because every element of 1st array has to be compared with every element of the 2nd array.

This in case, the 1st algorithm with O(n^2) does ***worse*** than the 2nd algorithm with O(nlogn).

**Question 3: 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?**

From an infrastructure perspective, using parallel computing or distributed computing over various cores can help reduce the time taken for computation.

Also from a code library perspective, using libraries like Numpy, is a lot efficient than using a List in python. Numpy serves as a wrapper around C and Fortran which is a lot faster than higher level languages like python. Also, in a NumPy array, the elements are stored sequentially, whereas in a List, arrays are stored in random locations in the memory, which makes working with an array faster.