**CS202 – HW1 REPORT**

**QUESTIONS 1 AND 3**

**Olcay Akman**

**21702671**

**Question 1**

1. The given functions sorted in increasing order of their asymptotic complexity is as follows:

(f5(n) = n1/logn) < f4(n) = logn < f10(n) = n1/2 < (f9(n) = log(n!)) = (f6(n) = nlogn) < f8(n) = n3 < f2(n) = nlog(logn) < f7(n) = en < f1(n) = 10n < f3(n) = n!

Because:

f1(n) = 10n  = O(10n)

f2(n) = nlog(logn) = O(nlog(logn))

f3(n) = n! = O(n!)

f4(n) = logn = O(logn)

f5(n) = n1/logn = O(1)

f6(n) = nlogn = O(nlogn)

f7(n) = en = O(en)

f8(n) = n3 = O(n3)

f9(n) = log(n!) = O(nlogn)

f10(n) = n1/2 = O(n1/2)

1. The average processing time T(n) of the following algorithm where random(n) takes ϴ(1) time should be as follows:

int test (int n) {

if (n <=0)

return 0;

else {

int i = random (n);

return ( test (i) + test (n -1-i));

}

}

It takes constant time to find a random number and assign it to i. It also takes constant time to do the comparison in the if statement. It takes constant time to “return 0;”. Thus, all these can be combined to be represented as in our final product for . Now only the return statement of the “else” part remains unresolved, in which the function itself is called again, resulting in recursion.

In order to calculate the average processing time, we need to deduce the value of i on each iteration of the function. On average, i should be .

*.*

*.*

*.*

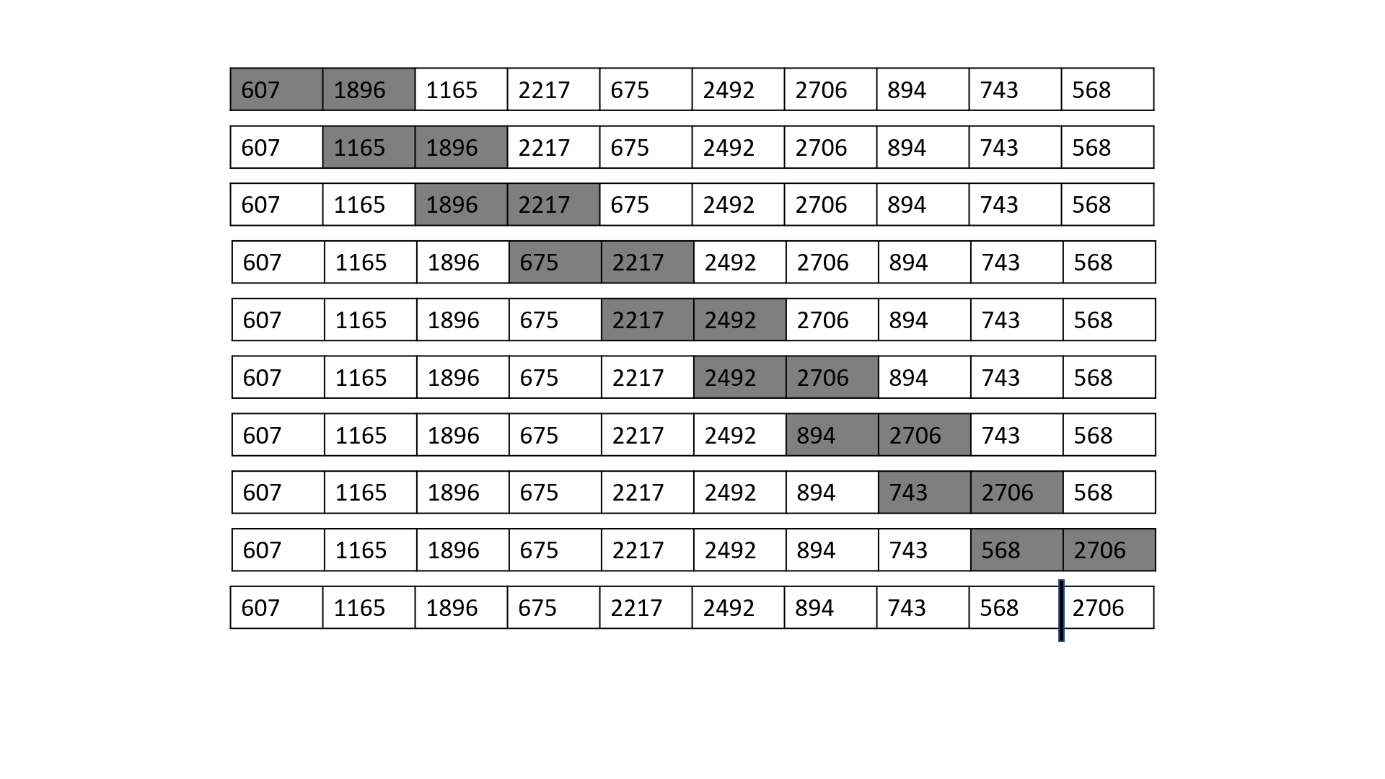
*i) + 2\*i\**

1. The array to be sorted is:

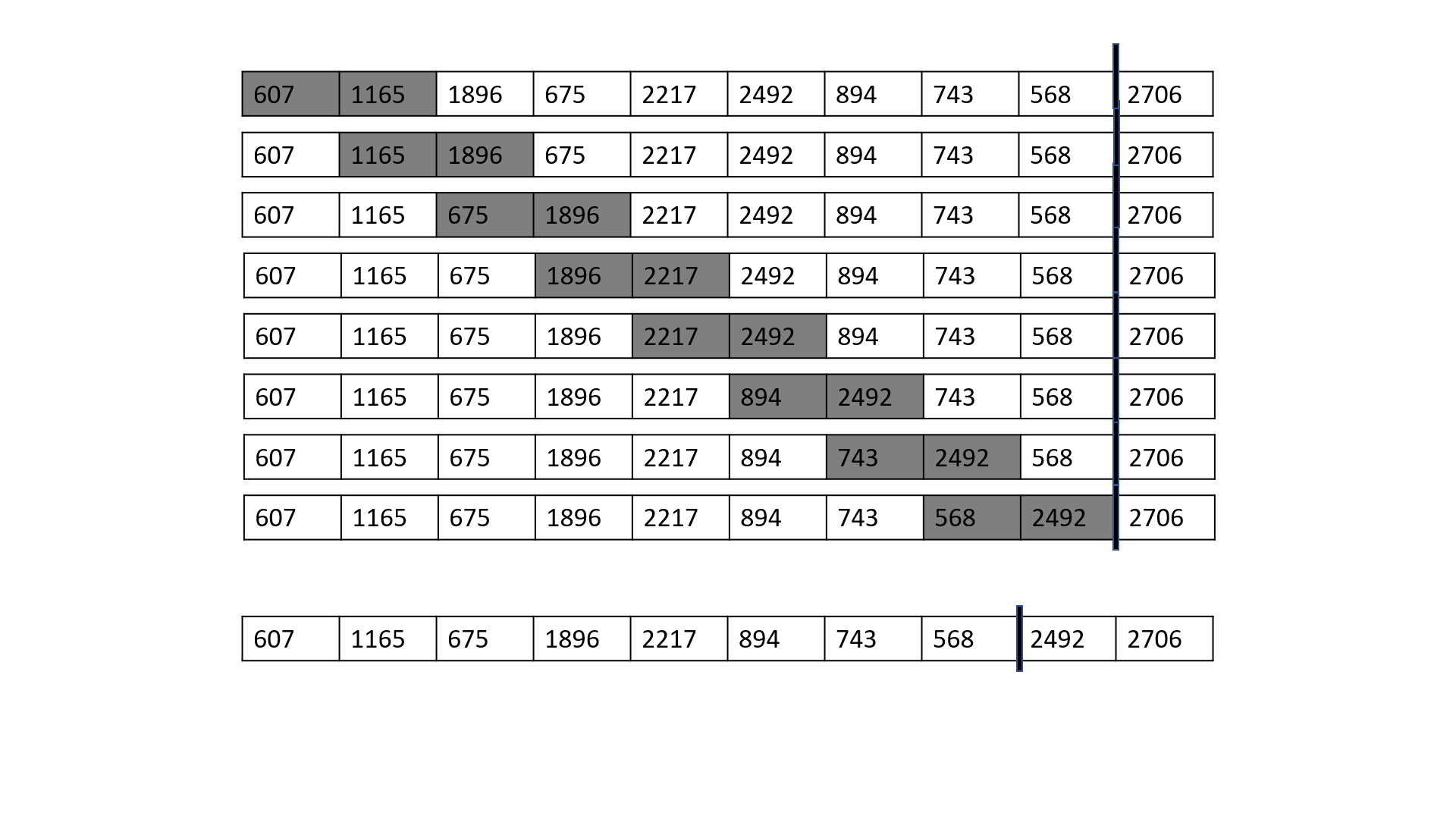
[607, 1896, 1165, 2217, 675, 2492, 2706, 894, 743, 568]

Let’s first apply Bubble sort to it:

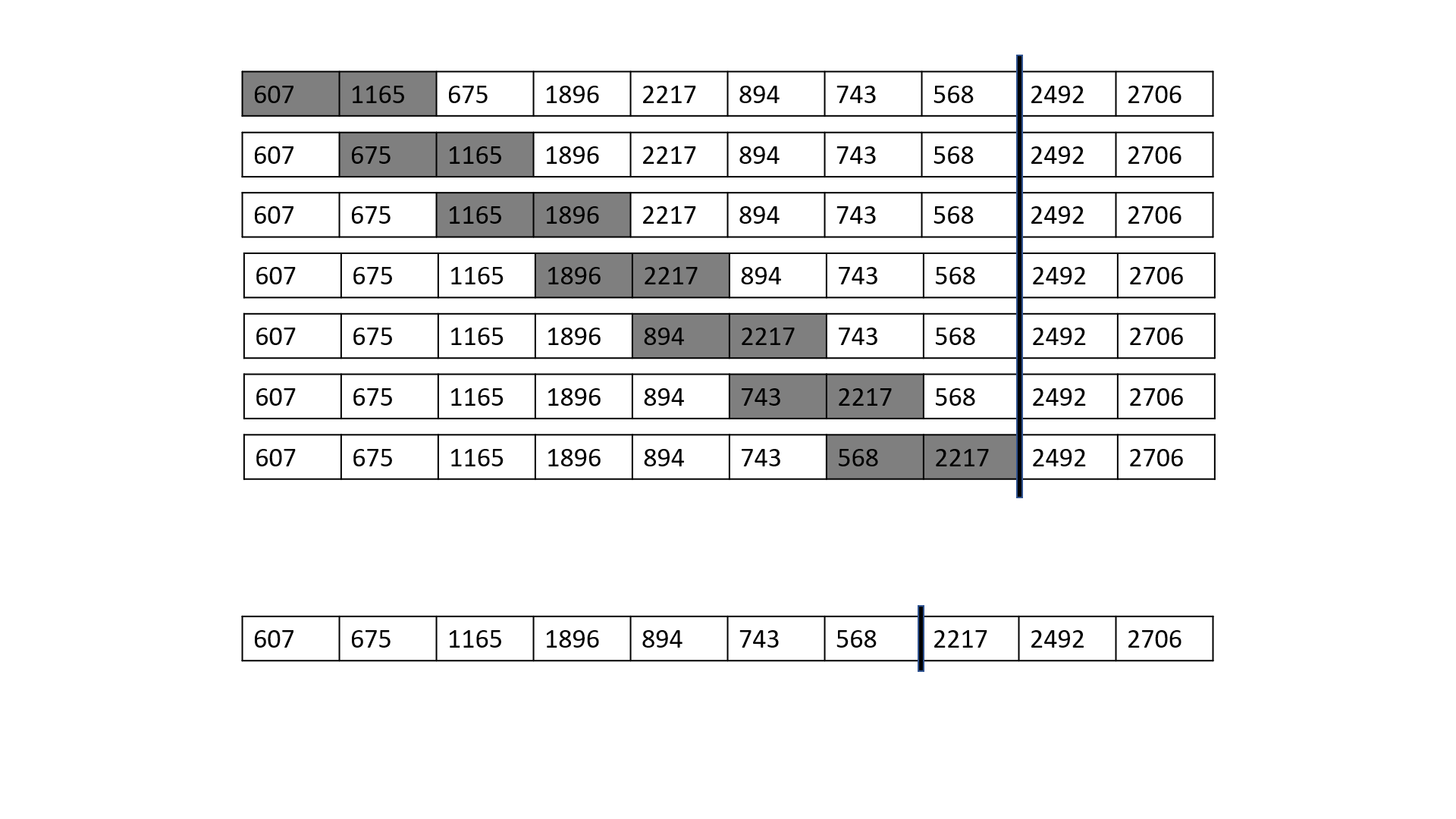
Pass 1:



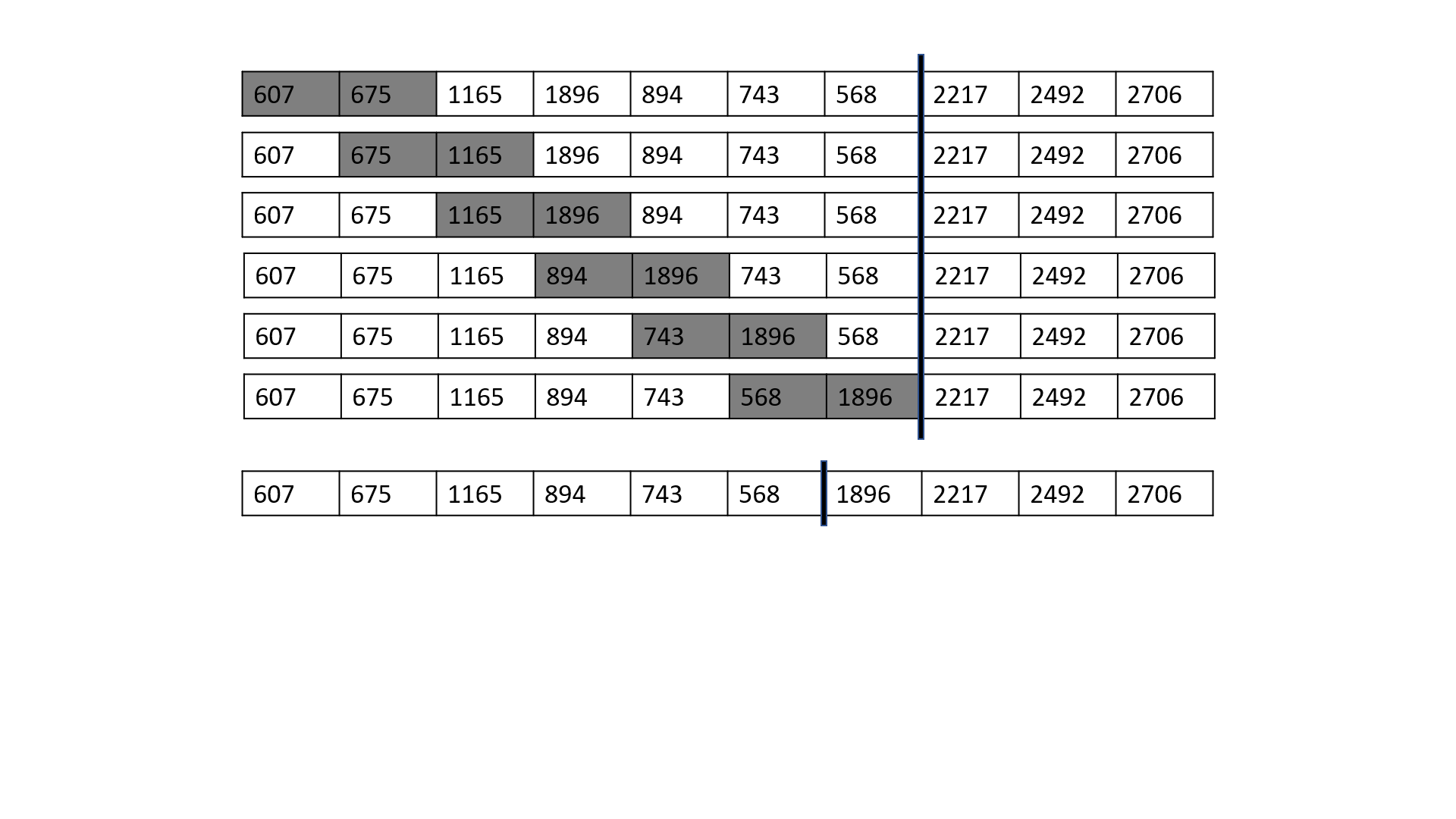
Pass 2:



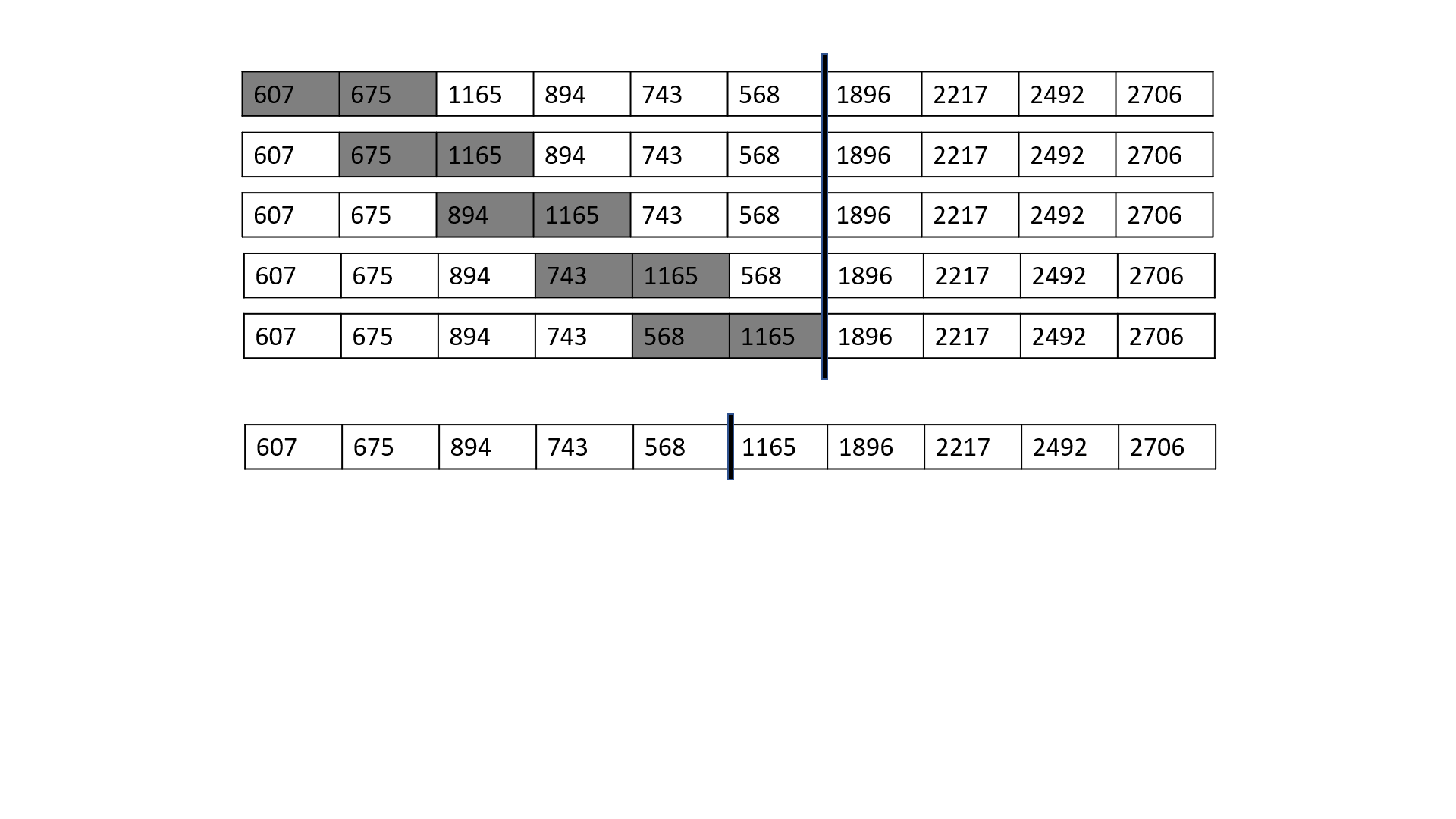
Pass 3:



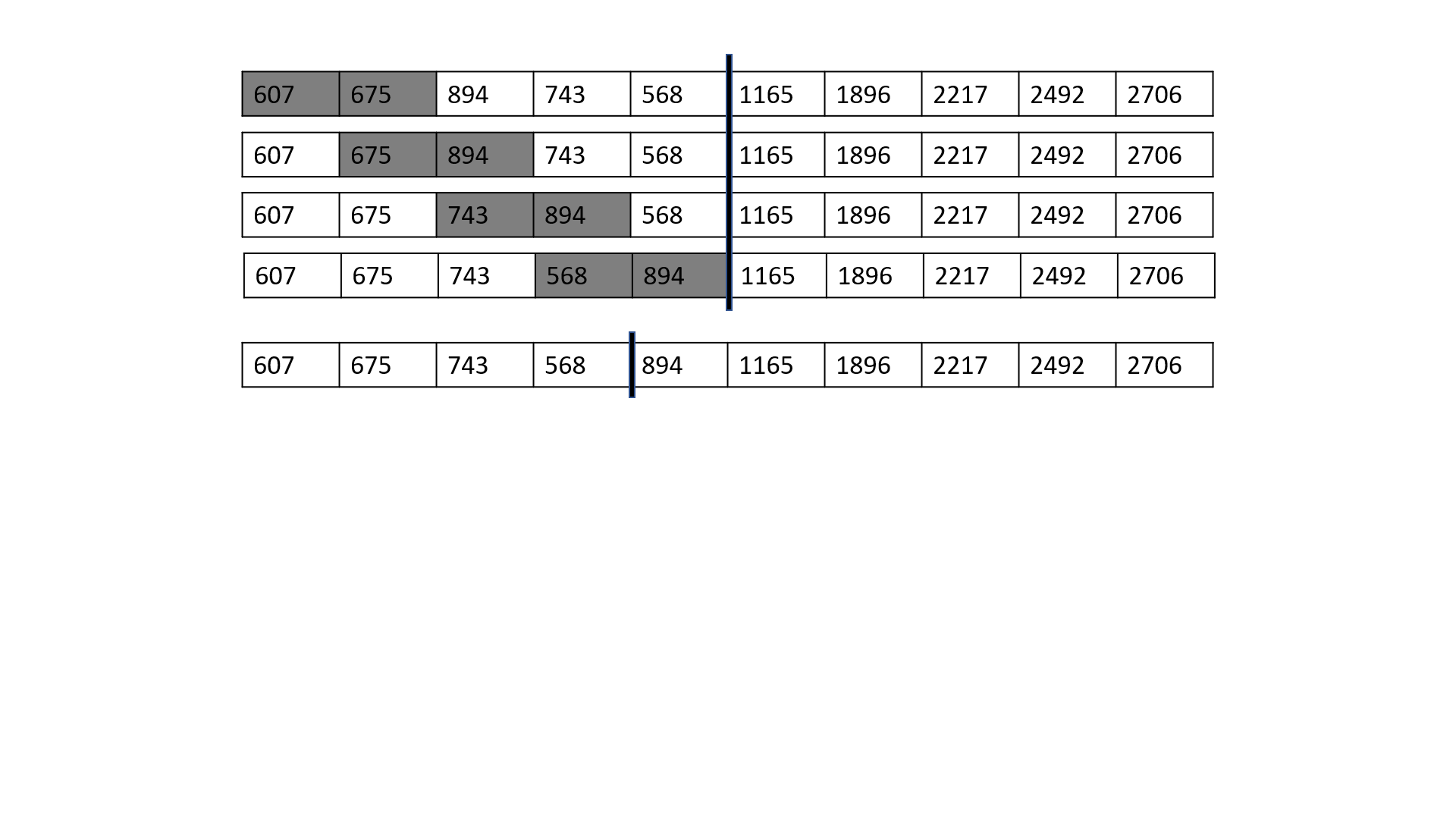
Pass 4:



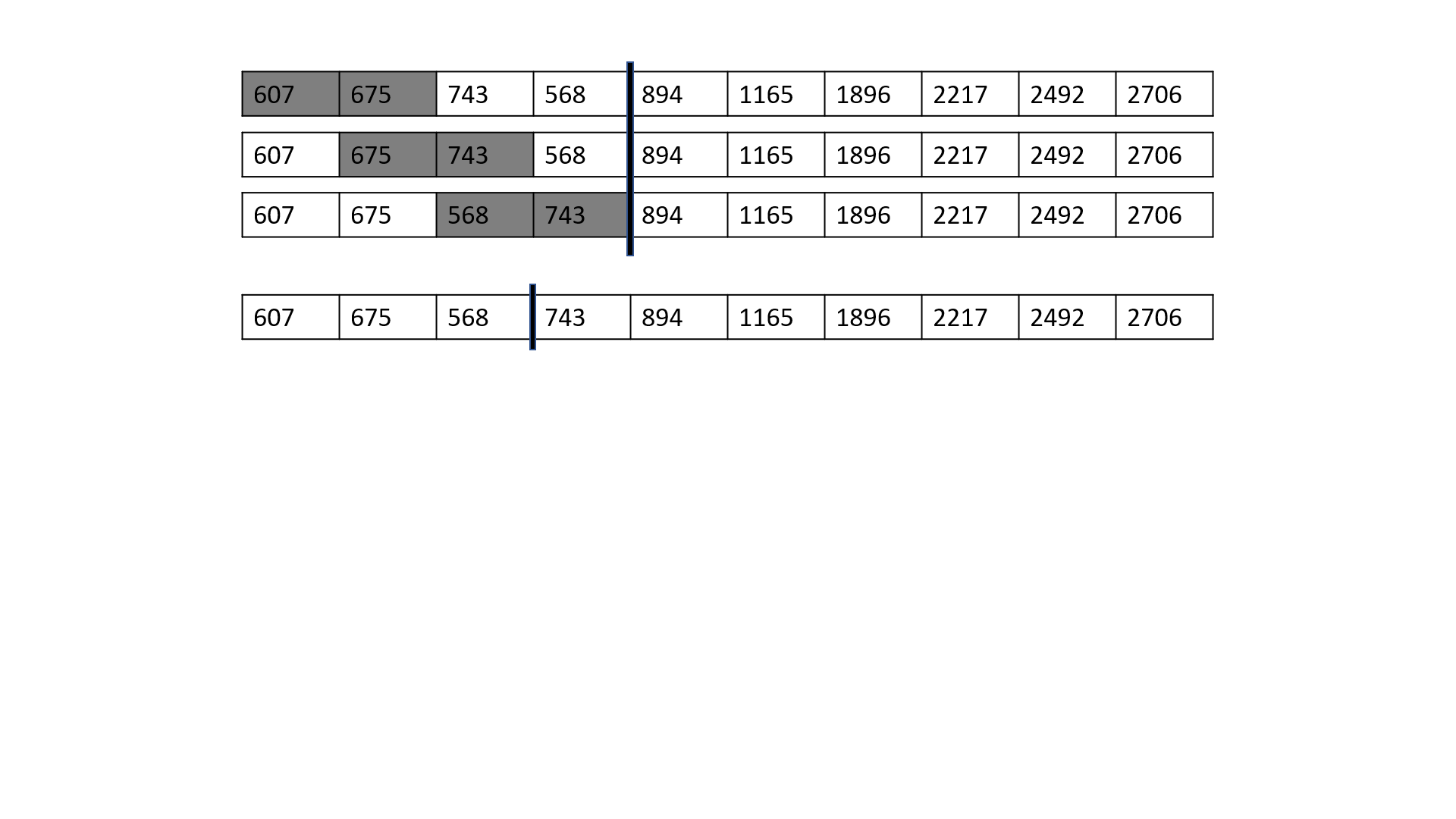
Pass 5:



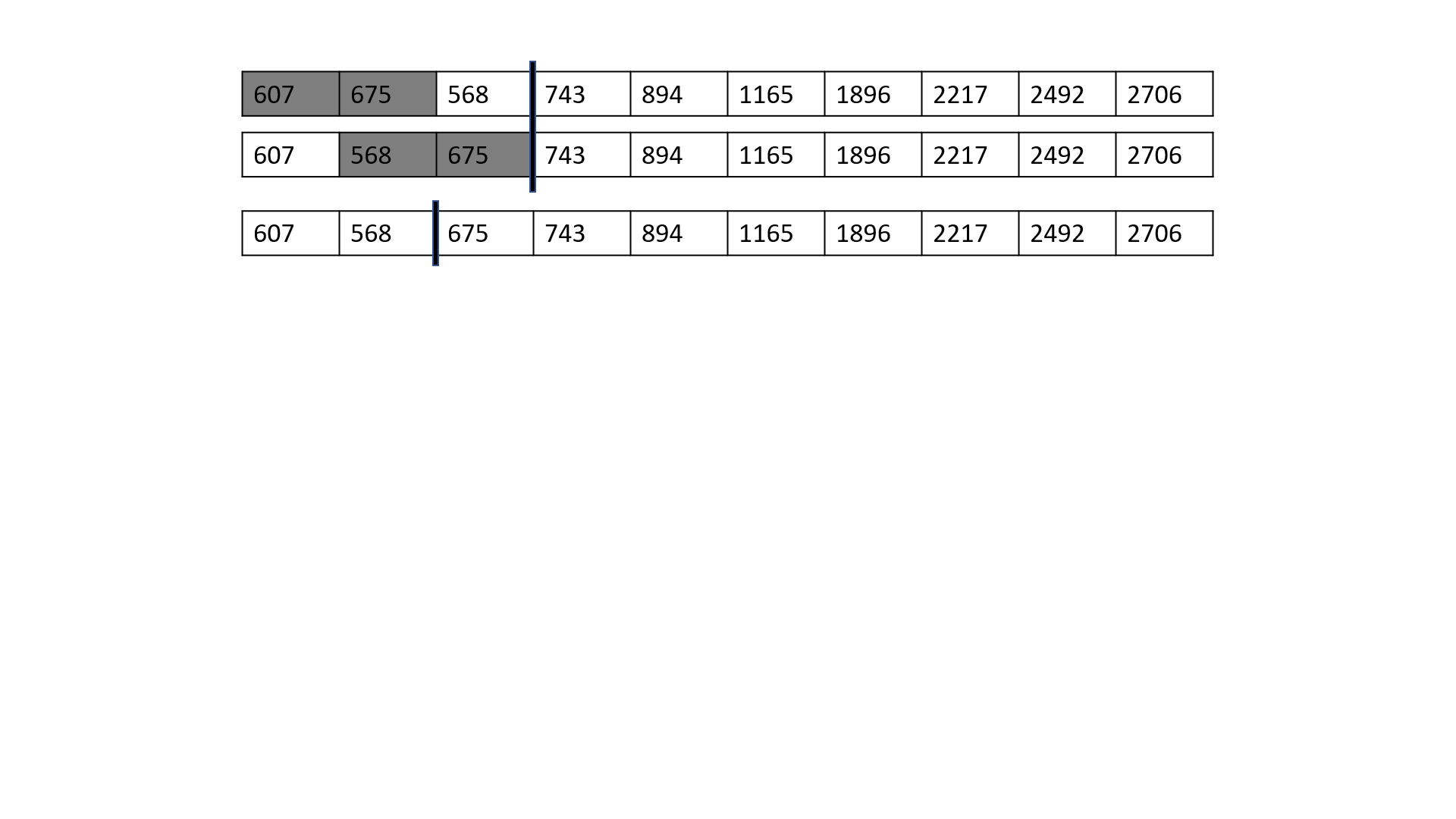
Pass 6:



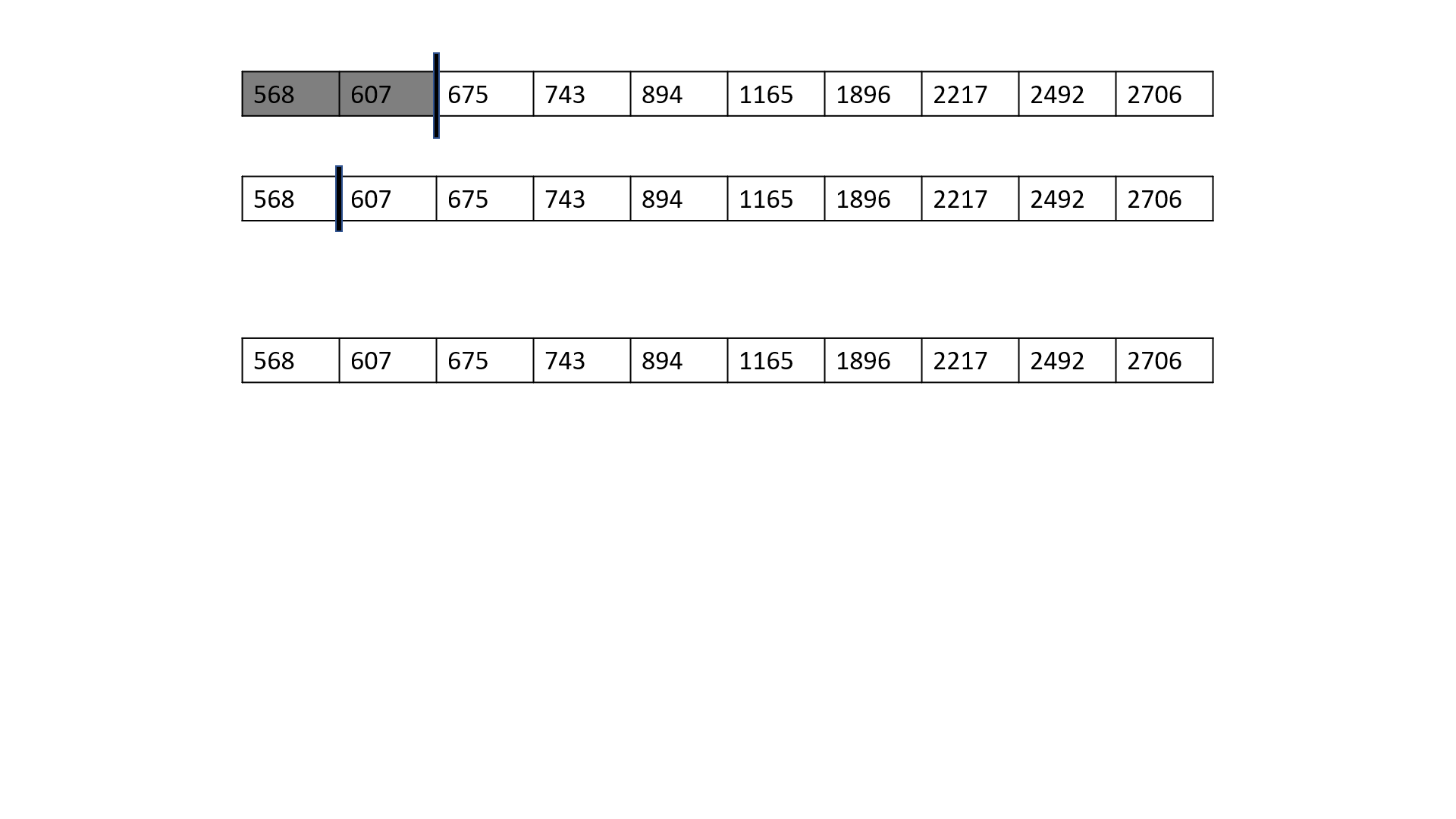
Pass 7:



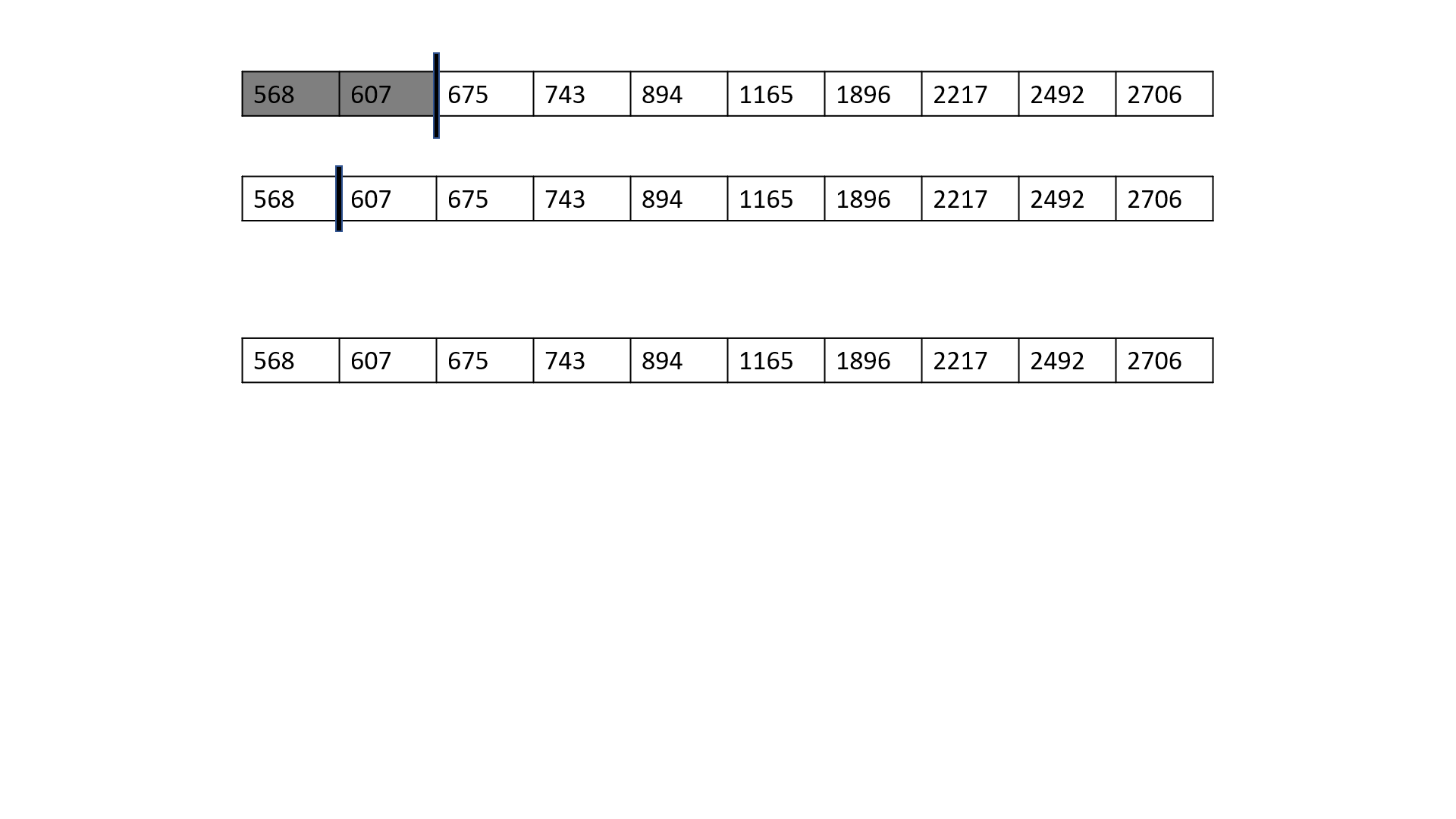
Pass 8:



Pass 9:

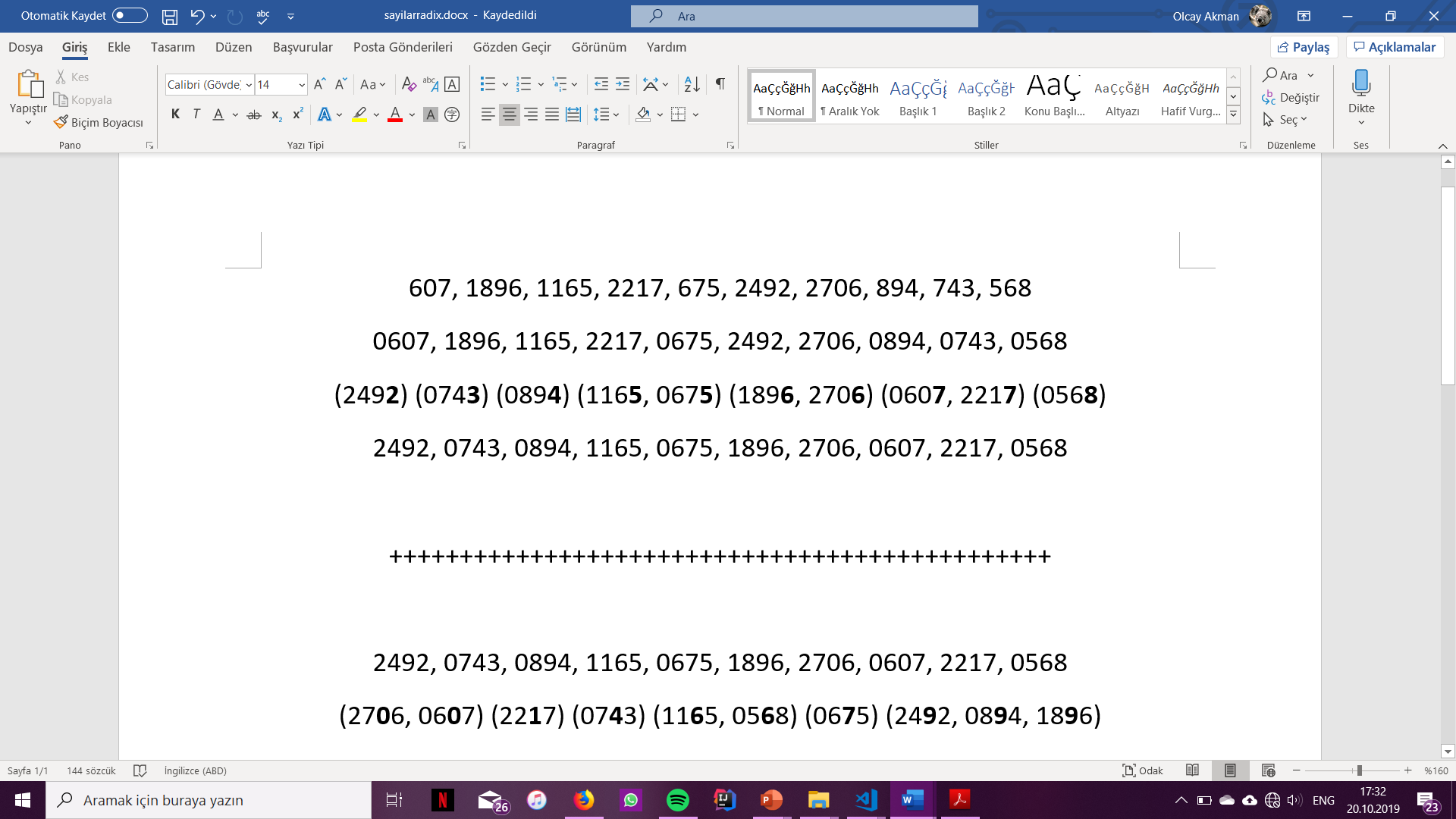


The sorted array:

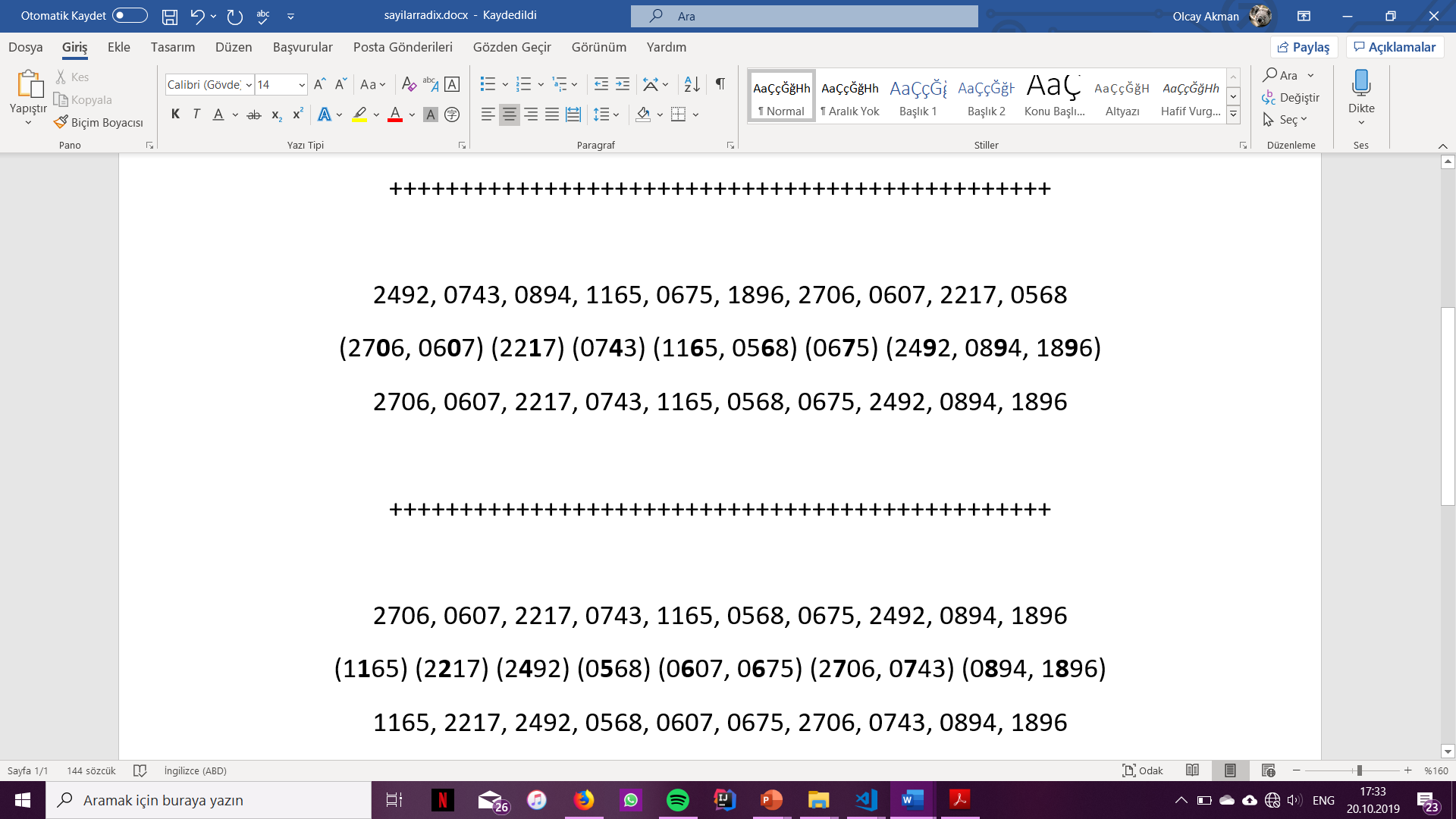


Similarly, the initial array will follow the following steps if we apply radix sort on it:

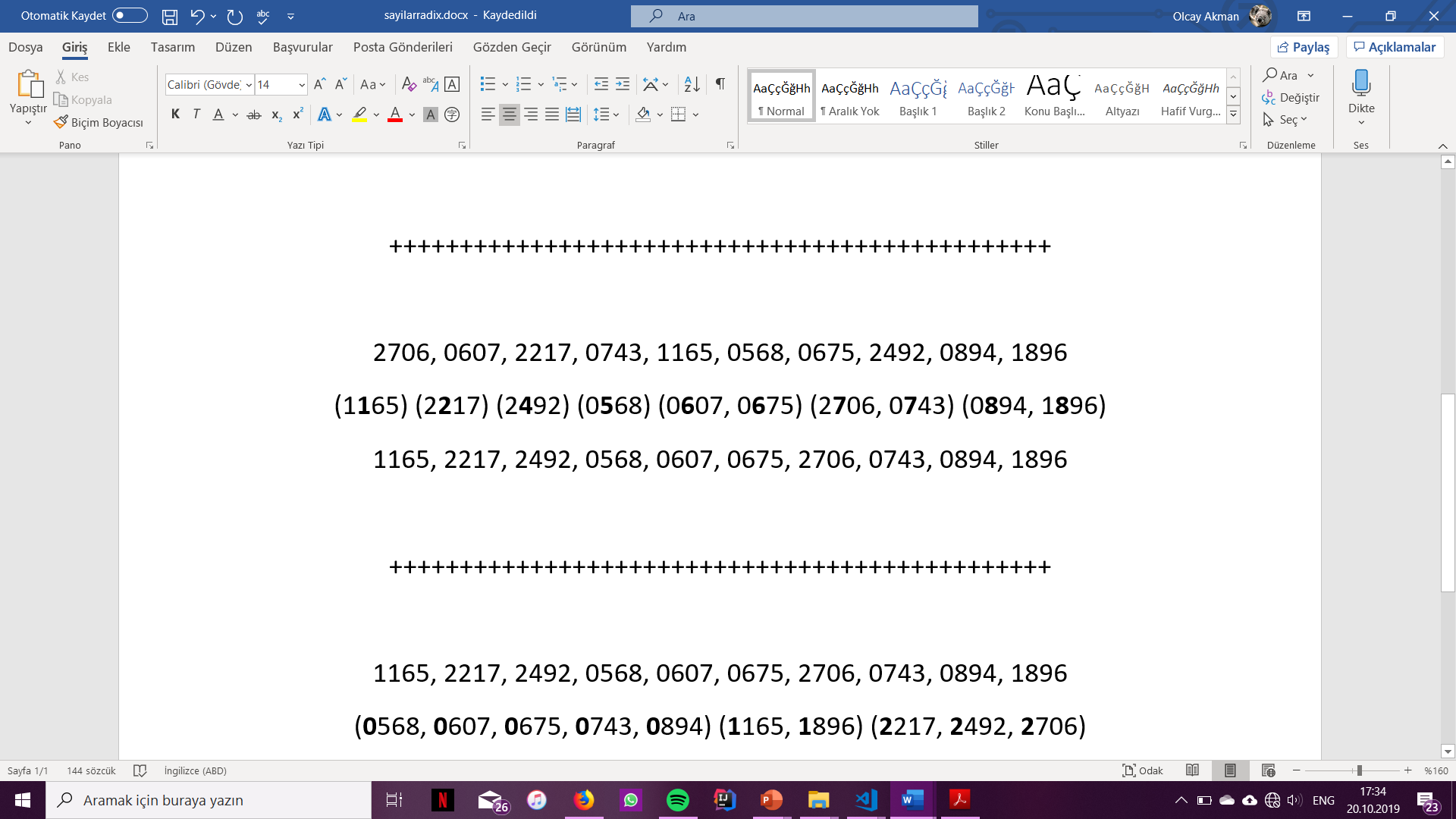
Sorting by 1st digits:



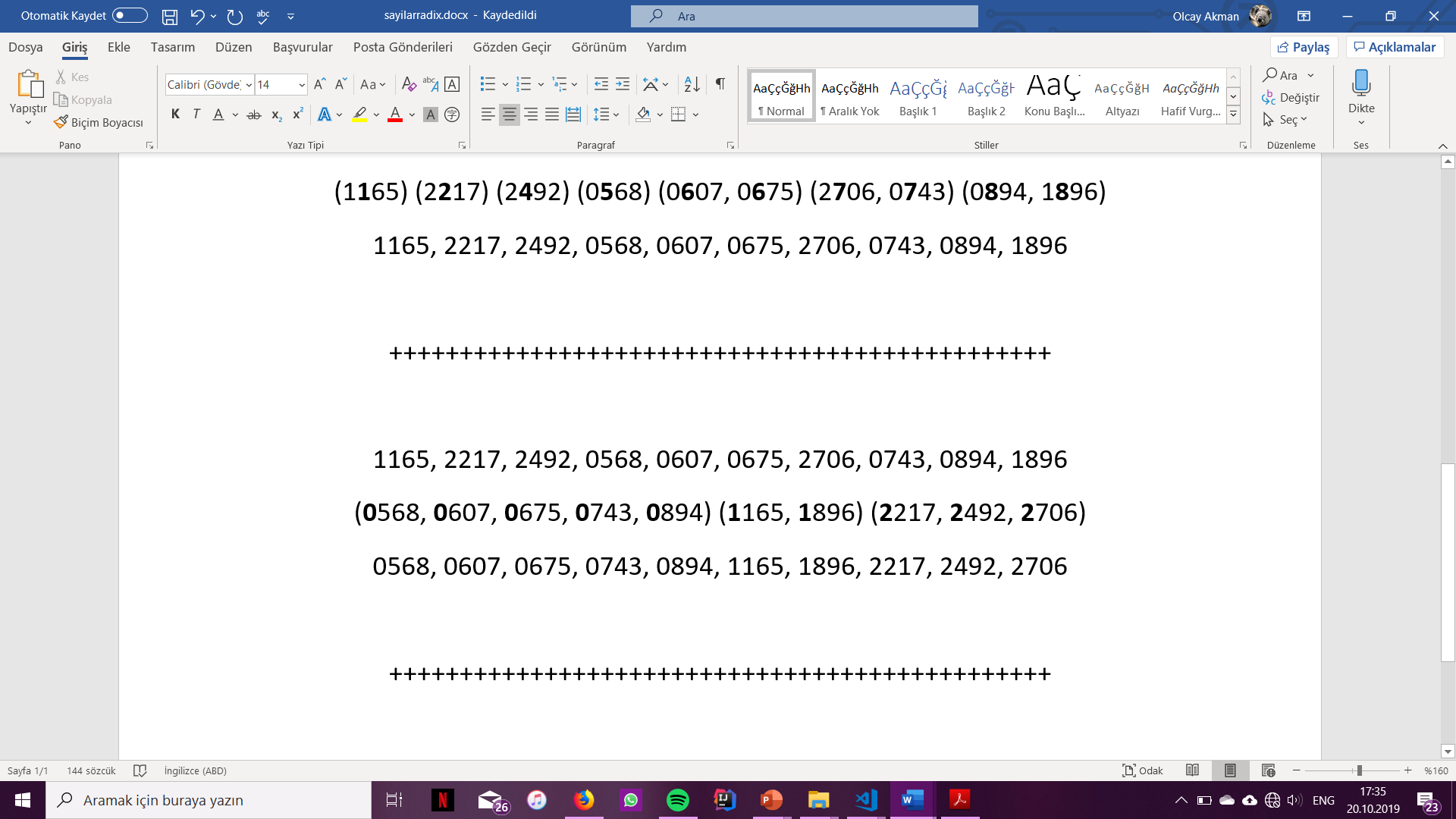
Sorting by 2nd digits:



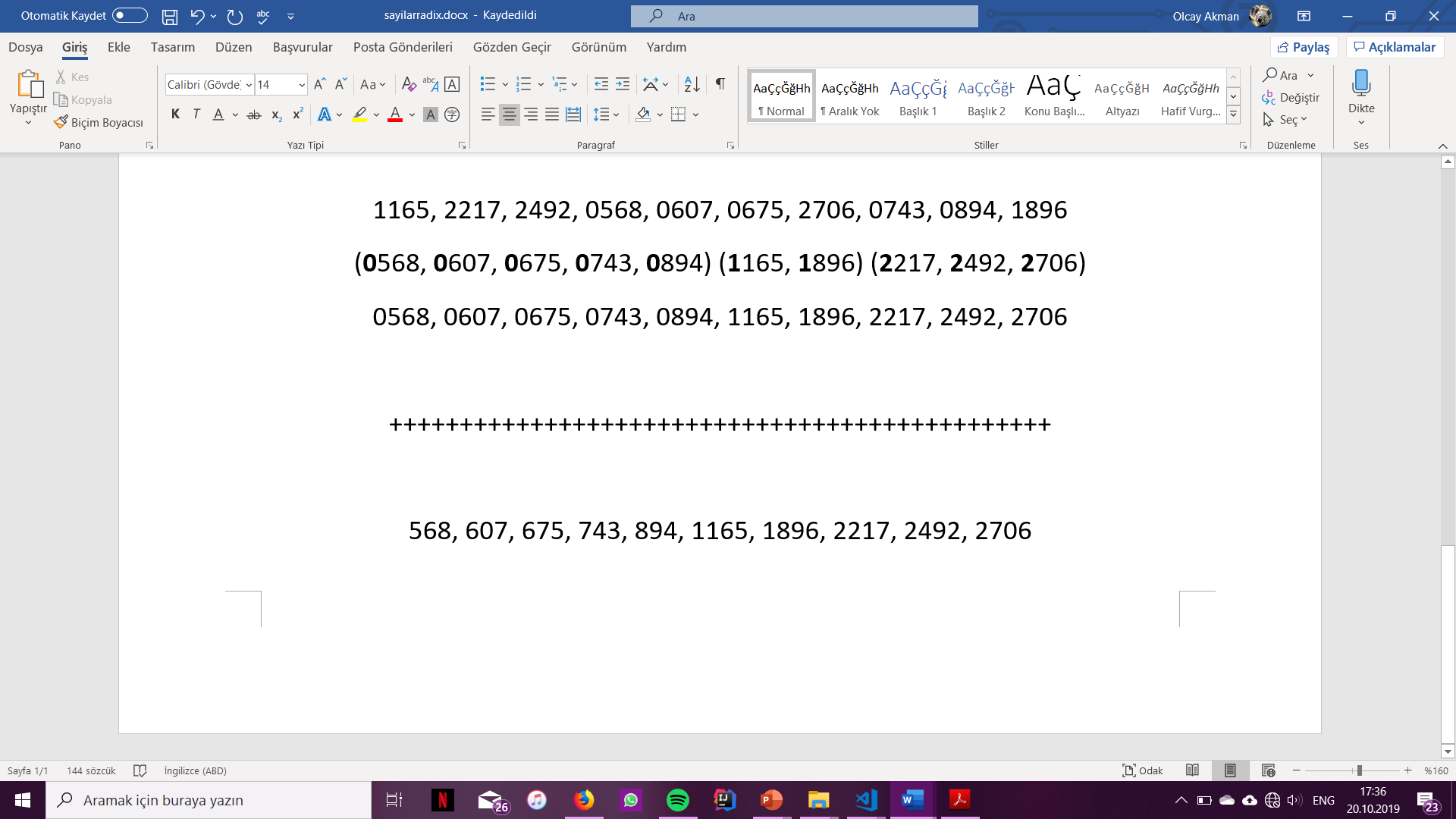
Sorting by 3rd digits:



Sorting by 4th digits:



The sorted array after radix sort:

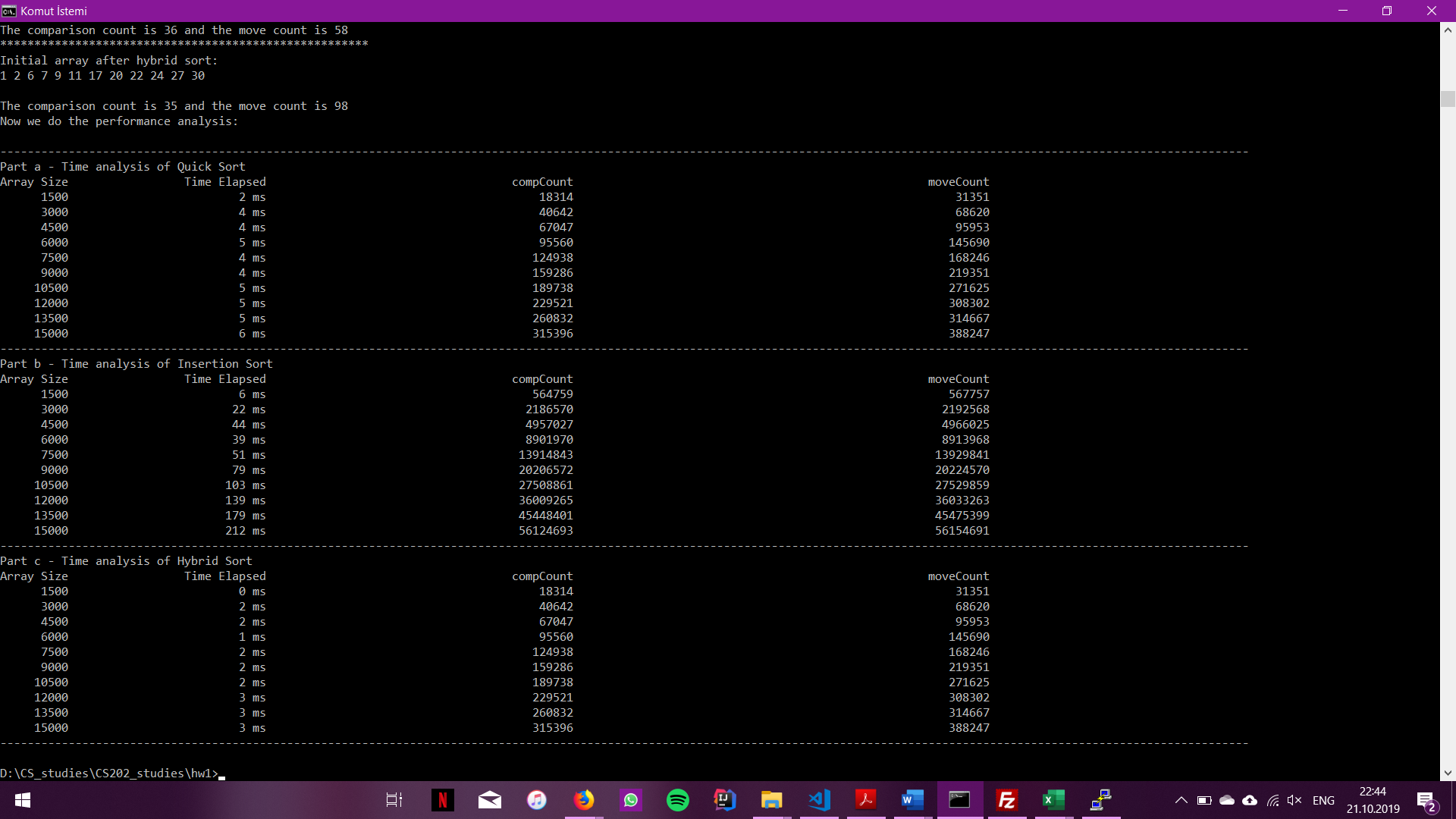


**Question 3**

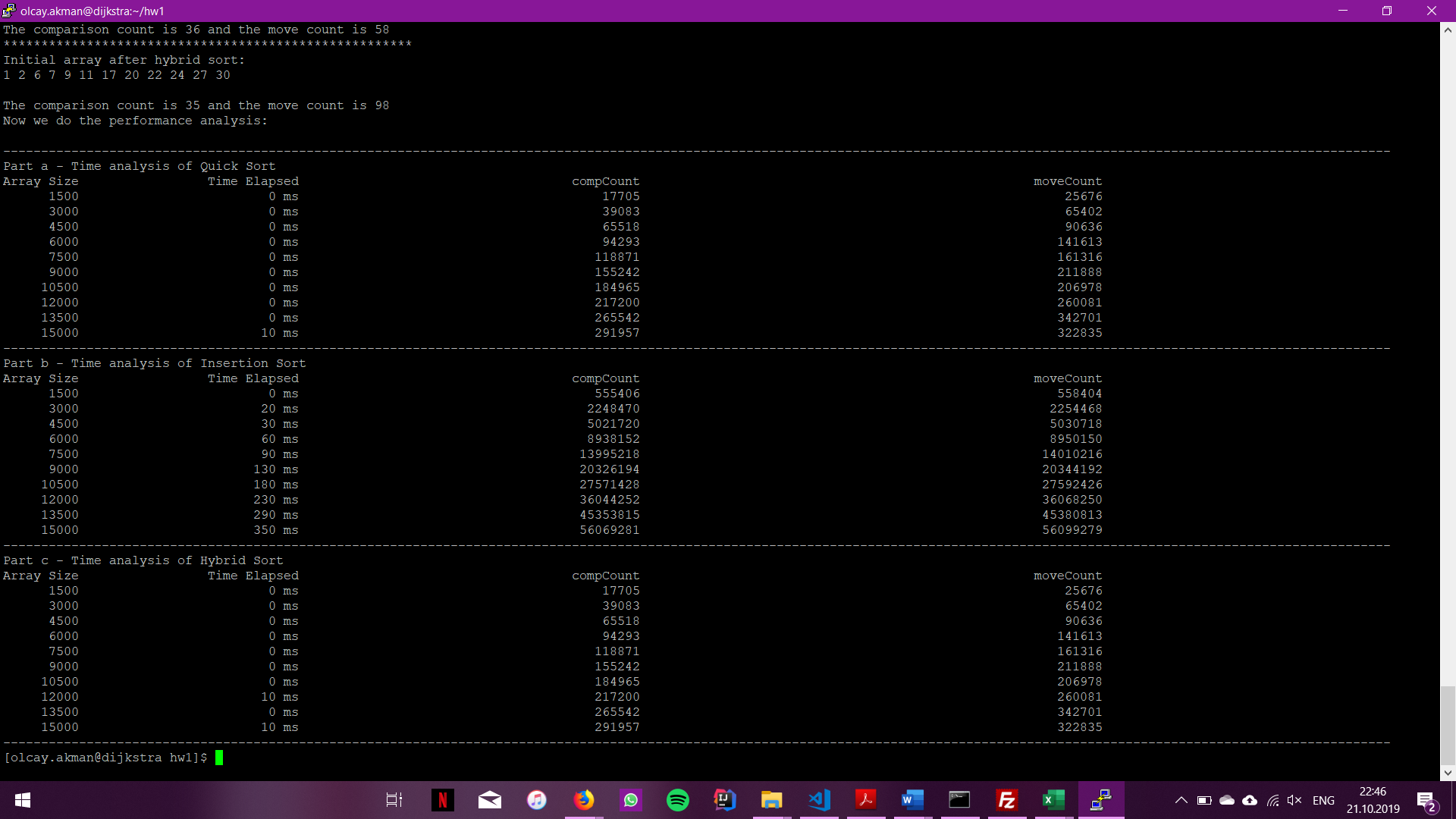
The elapsed time of each sorting algorithm varies from one run to another and it varies from device to device. In order to calculate the elapsed times of each sorting algorithm, in the performanceAnalysis method of question 2, a clock function from the C++ standard library was used, so that the elapsed times could be calculated with regards to the time of the computer, thus the variations from device to device. The variations from run to run in the same device depend on the processor of the computer being used at that moment.

After having run my code in my Windows machine and on the Dijkstra server, I have obtained fairly different results. I decided to rely on the physical machine at hand, that is my computer, to plot my graph. Below are the screenshots of what I have obtained from both the Dijkstra compiler and the compiler on my Windows machine.

Windows machine:



Dijkstra:



And below is my graph containing the data from the Windows machine:

Although rather invisible here, the quick sort and the hybrid sort functions take relatively similar times to be executed, whereas the insertion sort takes much longer to do so. Hybrid sort is the most efficient of the three for it utilizes both the quick sort and the insertion sort wherever they are swifter. That is, quick sort is fast on larger amounts of data whereas insertion sort is faster on smaller amounts of data. Since hybrid sort applies quick sort when array size is large, and uses insertion sort when array size is small, it is the most efficient of the three.