Exercise 1: Algorithm Design and Implementation.

1.) The algorithm takes an array a of integers, of length n, and returns s, an integer which is the SVD of a, or zero if no SVD is present. This algorithm uses nested iteration in the form of for loops to compare each element in the array to each other element, and if they are the same, increment a count.

After the element is compared to the other elements in the array, the count is checked to see it appears over n/2 times, and if it does it is returned as the SVD. Each element in the array is then checked in this way until either an SVD is returned and the algorithm ends, or an SVD is found and the algorithm ends by returning 0 to signify this.

There is no form of sorting with this algorithm nor is there any use of a secondary data structure.

2.) The algorithm takes an array a of integers, of length n, and returns s, an integer which is the SVD of a, or zero if no SVD is present. As set out by the coursework brief, this algorithm begins by using Java's Arrays.sort() method[1], which uses a Dual-Pivot Quicksort of O(nlog(n)) performance. This causes only a single iteration over the now sorted array to be required.

A pointer, i, is used to keep track of the current element which is then compared to the next element in the array. If they are the same, a counter is incremented and then the counter value is checked to see if it is an SVD. If the 2 elements compared are not the same, the counter resets to 1 to show a new value is present and the next comparison is started.

With the use of a Dual-Pivot Quicksort, secondary data structures are required.

3.) The algorithm takes an array a of integers, of length n, and returns s, an integer which is the SVD of a, or zero if no SVD is present. This algorithm is an implementation of Boyer-Moore's majority voting algorithm, which gives linear time complexity and constant space as no secondary data structures are needed.

The algorithm is implemented in two parts, the first finds a candidate for the SVD by effectively finding the majority element in the array. The index of the element that is currently considered to be the majority element is held, until the counter reaches zero which signifies there is an integer that appears an equal or greater number of times to the one currently being compared, and in that case the index is updated to keep track of the new prime candidate. The second part then simply verifies that the candidate is the majority in the array, appearing over n/2 times.

[1] https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#sort(byte[])

Algorithm 1 QuadraticSVD(a, n) return(s)

Require: An array a of integers, of length n.
Ensure: s, an integer which is the SVD of a, or zero if no SVD is present.

```
1: for (i ← 0 to n) do
2:     count ← 0
3:     for (j ← 0 to n) do
4:         if a[i] == a[j] then
5:         count ++
6:         if count > n/2 then
7:             return s = a[i]
8: return 0
```

Algorithm 1 Analysis.

1. Fundamental operation.
   return s = a[i].

2. Assuming worst case.

3. for i ← 1 to n do
   for j ← 1 to n do
   fundamental operation.

$$t_{A1} = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} 1 \right)$$

$$\sum_{i=1}^{p} 1 = p$$

$$= \sum_{i=1}^{n} n$$

$$\sum_{i=1}^{p} a = a \sum_{i=1}^{p} 1 = ap$$

$$= n \cdot \sum_{i=1}^{n} 1$$

$$\sum_{i=1}^{p} 1 = p$$

$$= n \cdot n$$

$$= n^2$$

4. $t_{A1}(n) = n^2$

t_{A1} is $O(n^2)$, where n is the length of the array a.

Algorithm 2. LogLinear SVD (a, n) return (s)

Require : An array a of integers, of length n.
Ensure : s, an integer which is the SVD of a, or zero if no SVD is present.

```
1:  Arrays.sort(a)
2:  i ← 0
3:  count ← 1
4:  while (i < n - 1) do
5:      if a[i] == a[i + 1] then
6:          count ++
7:          if count > n / 2 then
8:              return s:a[i]
9:      else
10:         count ← 1
11:     i ++
12: return 0
```

Algorithm 2 Analysis.

1. Fundamental operation.
   return s = a[:]
2. Assuming worst case.
3. Java's Arrays class method Arrays.sort has time complexity with $O(n \log(n))$ time complexity.

   Arrays.sort(a) + while (i < n-1) do
   return s = a[:]

$t_{A2} = \sum_{i=1}^{n-1} 1$ $\quad \sum_{i=1}^{p} 1 = p$

$= n - 1$

with sort ...

$n\log(n) + n - 1$

constants can be discounted.

$n\log(n) + n$

4. $t_{A2}(n) = n\log(n) + n$

$O(n\log(n))$ is slower than $O(n)$ so at worst case ...

t_{A2} is $O(n\log n)$, where n is the length of the array a.

Algorithm 3. Linear SVD (a, n) return (s)

Require : An array a of integers, of length n.
Ensure : s, an integer which is the SVD of a, or zero if no SVD is present

```
1:  index ← 0                          // finds candidate for SVD.
2:  count ← 1
3:  for (i ← 1 to n) do
4:      if a[index] = a[i] then
5:          count ++
6:      else
7:          count --
8:      if count == 0 then
9:          count ← 1
10:         index ← i
11: svd_candidate ← a[index]
12: count ← 0                          // checks if SVD candidate
13: for (i ← 0 to n) do                   does occur > n/2 times.
14:     if a[i] == svd_candidate then
15:         count ++
16: if count > n/2 then
17:     return s ← svd_candidate
18: return 0.
```

Algorithm 3 Analysis.

1. Fundamental operation.

There are 2 seperate parts to this algorithm, so 2 fundamental operations are identified.

① index ← i (first loop)
② count ++ (second loop)

2. Assuming worst case.

3. ① for (i ← 1 to n) do
index ← i

$$t_{A3} = \sum_{i=1}^{n} 1$$

$$\sum_{i=1}^{p} 1 = p$$

$$= n$$

② for (i ← 1 to n) do
count ++

$$t_{A3} = \sum_{i=1}^{n} 1$$

$$\sum_{i=1}^{p} 1 = p$$

$$= n$$

$$t_{A3} = ① + ②$$
$$= n + n$$

4. t_{A3} is $O(n)$, where n is the length of the array a.

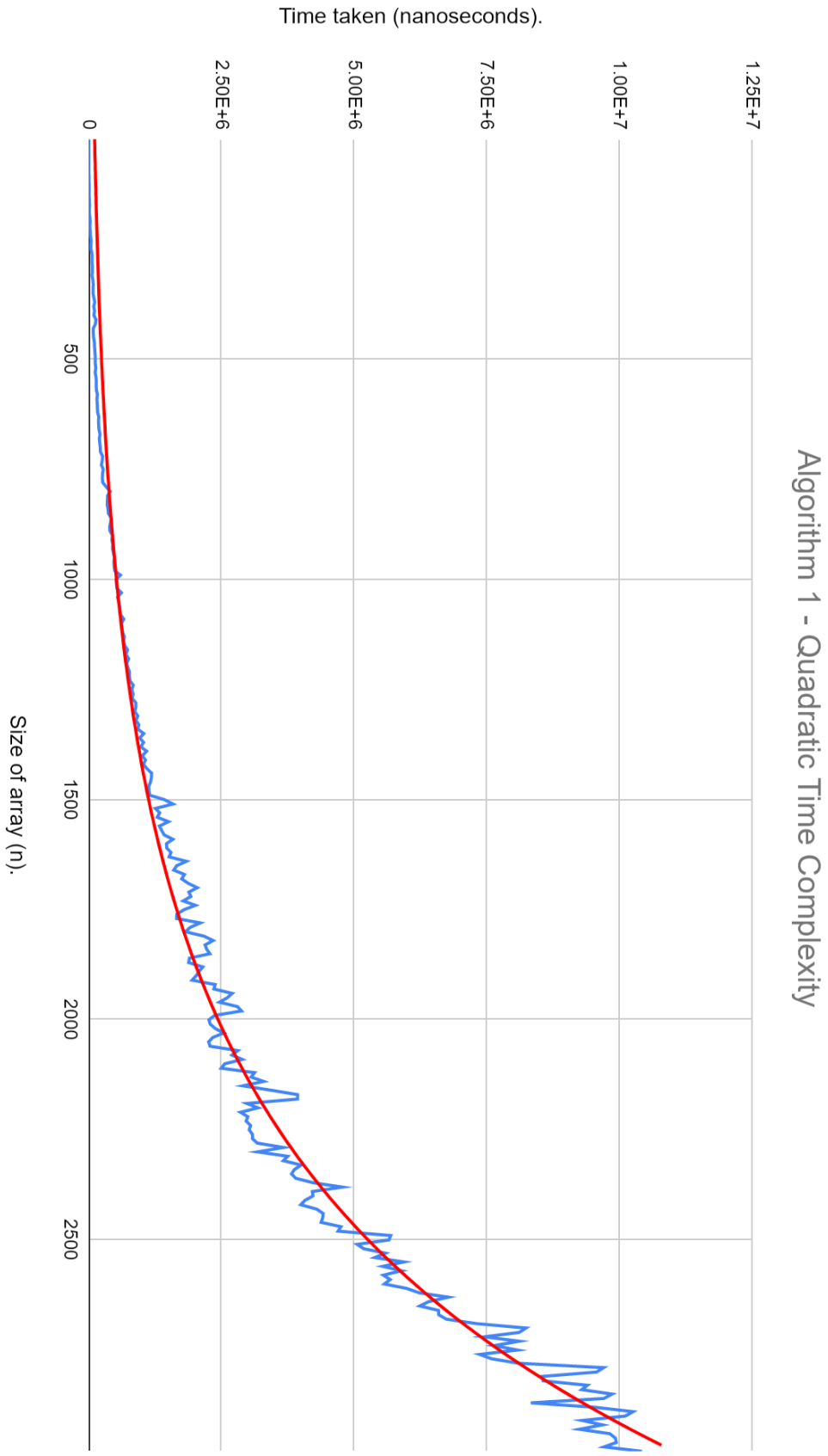
Algorithm Timing Experiment Results

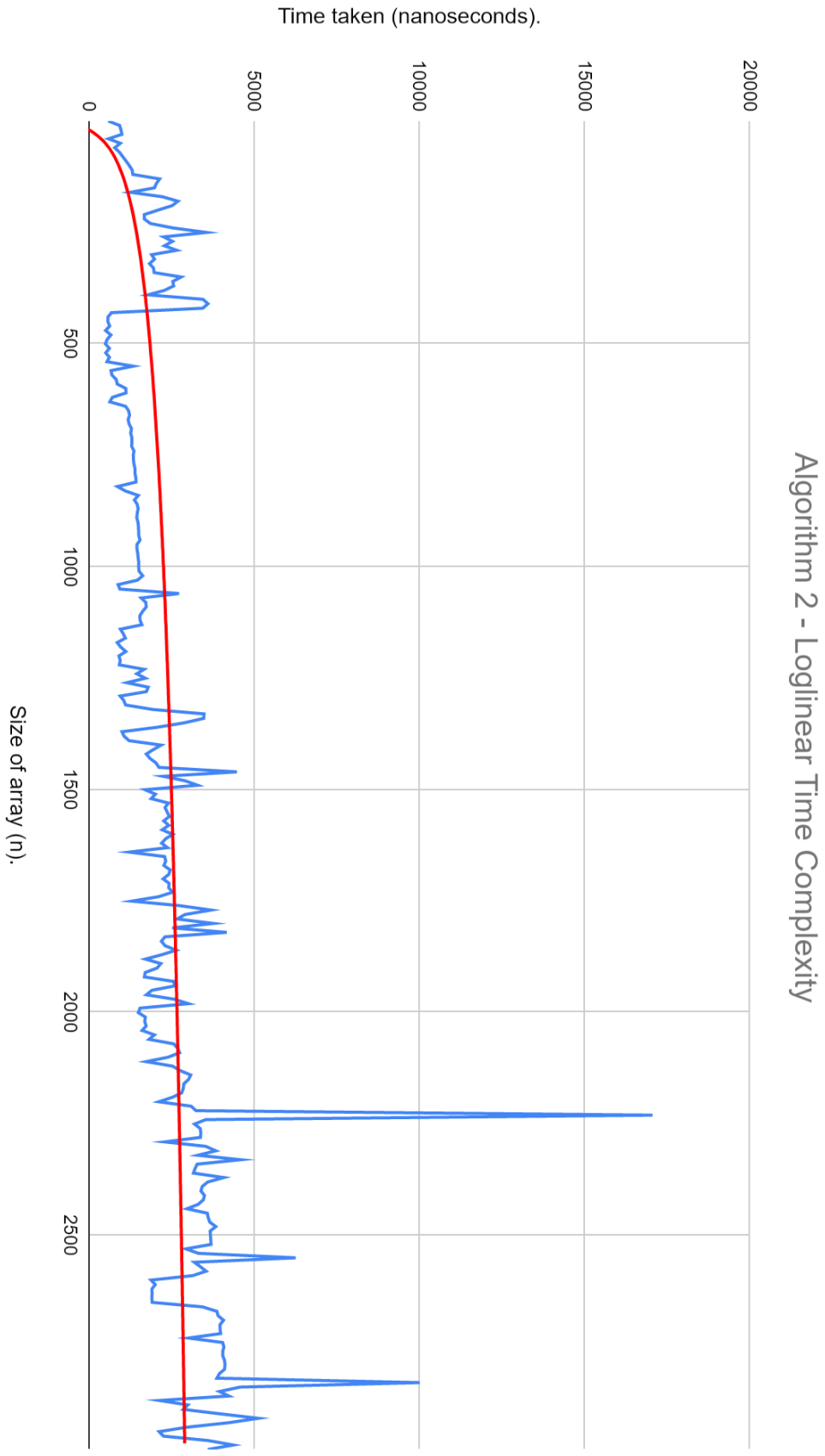
My method involved testing each of my implemented algorithms by increasing the size of n, and recording the amount of time it took for the code to successfully complete.

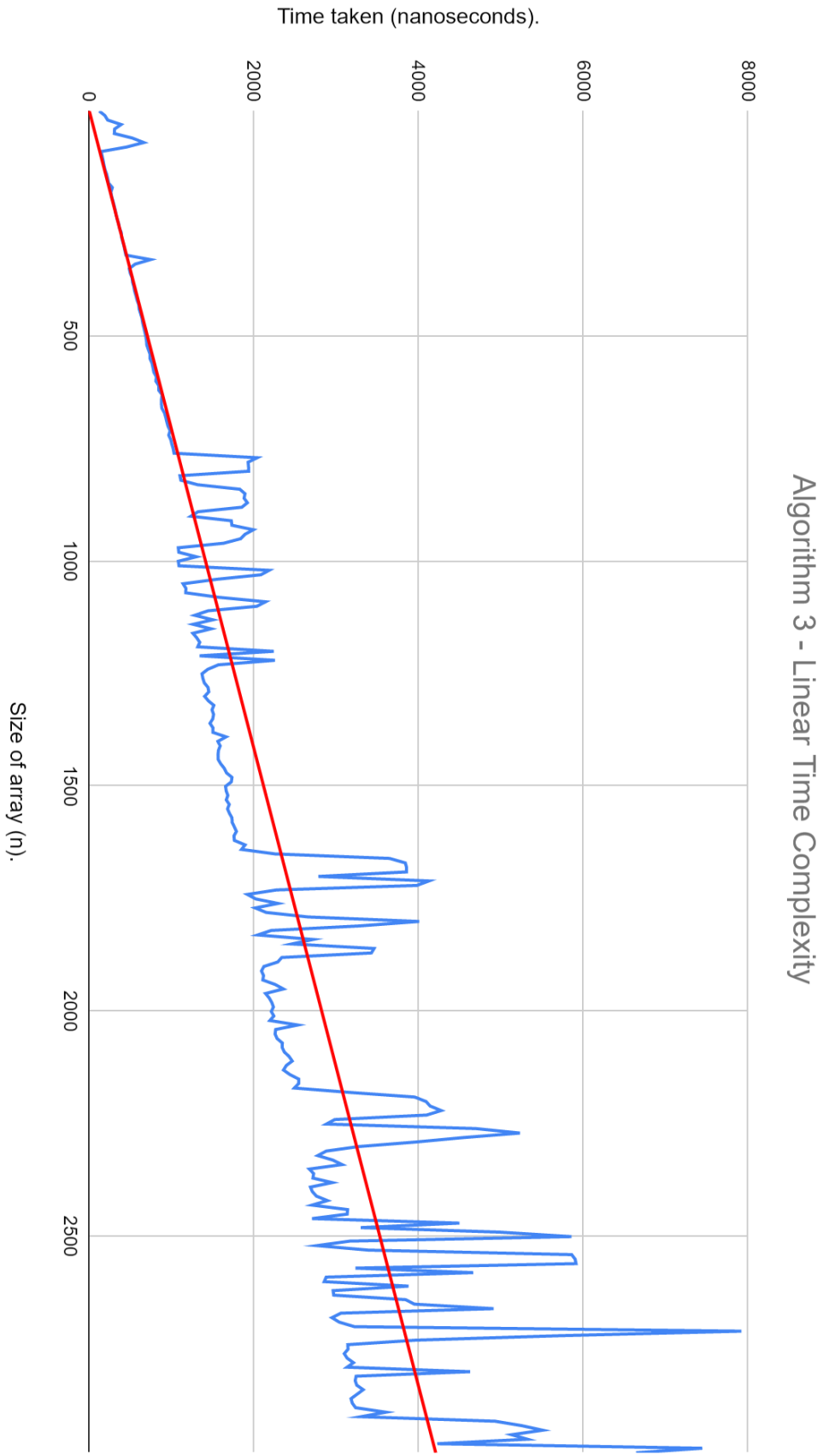
The independent variable, n, was increased from 1 to 3000 in increments of 10, and each reading was repeated 500 times and an average was calculated for the time taken for each value of n.

The following pages show the graphed results for each of the 3 algorithms.

A link to the numerical results can be found here
.
https://docs.google.com/spreadsheets/d/16a91cRWaAIIGjOkuh5CxbvF-6FAoLCGe14xxlmGqysk/edit?usp=sharing


Algorithm 1 - Quadratic Time Complexity
Time taken (nanoseconds).
0
2.50E+6
5.00E+6
7.50E+6
1.00E+7
1.25E+7
Size of array (n).
500
1000
1500
2000
2500


Algorithm 2 - Loglinear Time Complexity
Time taken (nanoseconds).
0
5000
10000
15000
20000
Size of array (n).
0
500
1000
1500
2000
2500


Algorithm 3 - Linear Time Complexity
Time taken (nanoseconds).
0
2000
4000
6000
8000
Size of array (n).
0
500
1000
1500
2000
2500