CAB301 Algorithms and Complexity

Assignment 1

Empirical Analysis for Algorithms

Student name: Eliot Wilson n9175504

Submission Date: 22nd April 2016

1. Algorithm Description – Insertion Algorithm

For this assignment, the algorithm analysed was the insertion algorithm (See Appendix 1). An insertion algorithm reconstructs a given array of integers in ascending order one item at a time by re-arranging the variables using a temporary position. Initially, the algorithm assigns the first element in the array to a temporary position and creates a temporary value based one lower than that value. The algorithm uses two segmented loops and a comparison statement to see if any value in the array is greater than the initial value **(Muhamma, 2010)**. While there are still greater values than the initial value, the algorithm adjusts the positioning of larger elements and re-inserts the initial value back into the list. This process is repeated incrementally for each element in the array.

2. Expected Case Efficiency

2.1. Basic Operation

A basic operation within a searching algorithm such as the Insertion Algorithm is often a section of operation that compares two or more values. Within the given algorithm, it can be determined that there is one basic operation.

*while(tempvalue >= 0 && **numlist[tempvalue] > value**){*

It can be seen that there are two comparative operations contained within the statement, however, only one of these operations is a comparison between variables and required to sort the arrays (highlighted above) and will therefore act as the basic operation.

Within the algorithm this comparison is assumed to run for every unordered element during the loop. Using this it is possible to determine the average efficiency case of the algorithm.

Average Efficiency - C $_{\text{Average}}$ (n^2)

The efficiency of the algorithm would be based on the instances in which the basic operation is met. This would mean that if the array had previously been sorted, or is empty, then the case scenario would be a linear relationship of **Θ(n) or Zero Operations**, however if the elements require searching and segment through the loop, then the case would be a quadratic relation as the required loops would be run for each instance. Similarly, the average efficiency would see the inner loop running on multiple elements and would therefor fall under the same efficiency class. i.e it would be bound by a Quadratic rate of growth, **Θ(n^2)** and would be closer to the worst case scenario in terms of run-time efficiency.

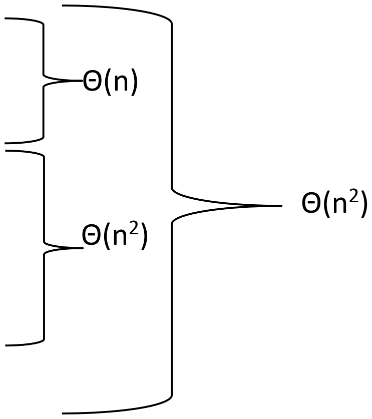
This is further re-enforced by Levitin **(2007)** who has found that, on average, the operations are executed ***n^2* / *4*** *times*. The average stated can be seen to have the same order of growth as the average efficiency calculated for the algorithm.

3. Experimentation Methodology

3.1.Algorithm Implementation

In order to accommodate the analysis of the algorithm, the code was constructed in C++ using the CodeBlocks environment as shown below.

```
for(int i = 1; i < Length; i++ ){
      value = numlist[i];
      tempvalue = i-1;
       while(tempvalue >= 0 && numlist[tempvalue] > value){
                 numlist[tempvalue+1] = numlist[tempvalue];
                  tempvalue--;
       }
        numlist[tempvalue+1] = value;
```



C++ code of the completed algorithm

The Insertion Algorithm was created in c++ but was then altered to include methods for recording the number of basic operations (See Appendix 4), as well as the execution time for the code relevant to the algorithm. i.e. the times taken in order to produce random integer arrays and the print statement times were not recorded. Furthermore, as print statements would reduce the efficiency of the processing algorithm, they were only used for testing the functionality and removed for the experimentation phases.

The code was run on two separate Windows laptops in order to test for any differences in processing speed that may affect the results. Due to the computers being of similar processing power, there was little difference in the results. The programs running in the background were also closed in order to minimise processing errors.

The recorded data was split into two series; one series for the elapsed execution time and one series for the number of basic operations. The varying data series were then output to text files in order to allow for data mapping and tabled into spreadsheets using Microsoft Excel.

4. Results

4.1. Proof of Correctness

Before analysis of the algorithm could take place, the functionality of the above code was verified through testing with small pre-set arrays. The tests were run with the same intended purpose of the algorithm in order to prove that the functionality would be consistent for the experiments. For code used in testing stage see Appendix 3.

For the first test, an array consisting of the elements; 1, 3, 5, 7, 9, 2, 4, 6 and 8 was used. The test re-ordered the array and recorded the number of basic operations, resulting in the following output:

***Array = {1,2,3,4,5,6,7,8,9} with 19 Operations Performed**.*

To test if this is correct, the average case efficiency as set by Levitin was used to calculate the predicted number of operations:

$$Operations = \frac{n^2}{4}$$

$$= \frac{9^2}{4}$$

$$= \frac{81}{4}$$

$$Operations = |20| \text{ Rounded down from 20.25}$$

The number of found operations can be explained to be one fewer the projected number as the first integer is the smallest and would not require sorting.

This output implies that the code correctly follows the expected functionality for sorting arrays. However, more testing helped to indicate if the purpose is fully met.

To further test that the number of operations is accurate, a pre-sorted array was used that consisted of the same elements as the previous test. As there was no need for the sorting algorithm to function, the output stating the number of operations correctly reflects this change.

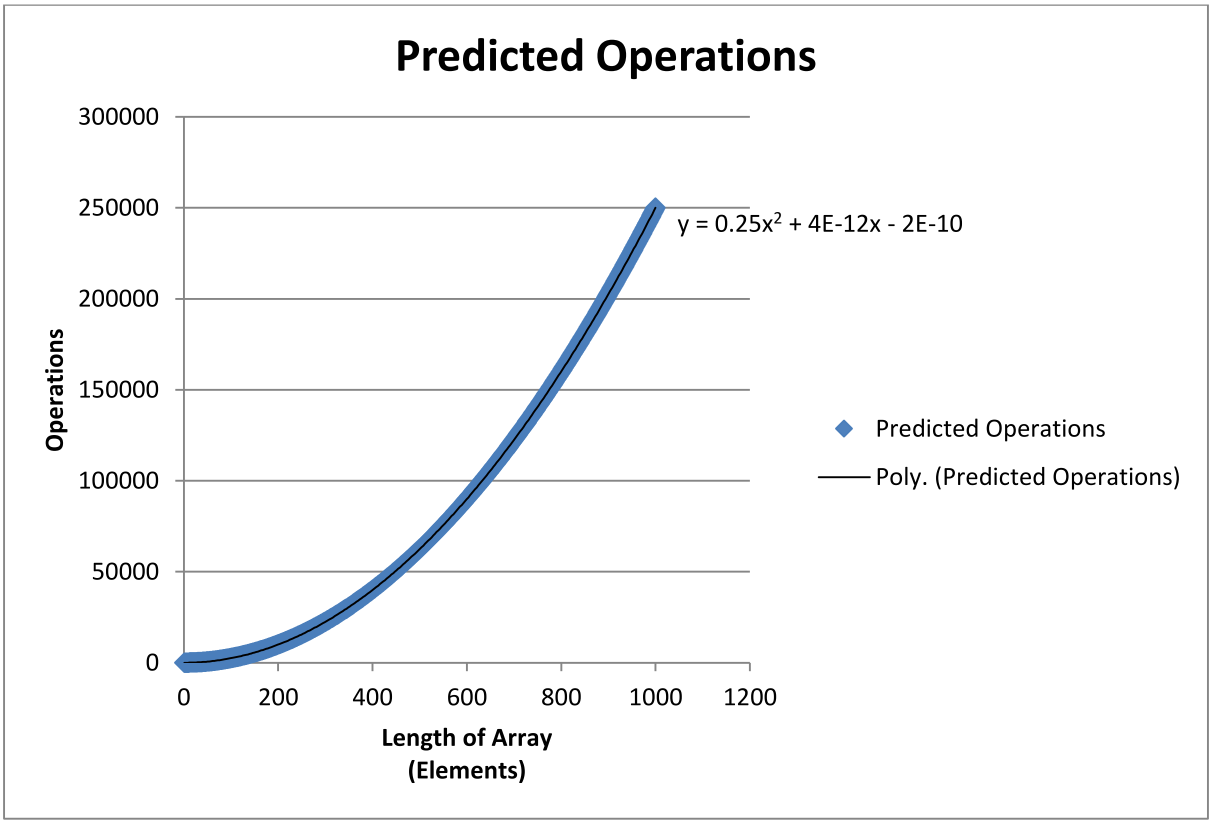
***Array = {1,2,3,4,5,6,7,8,9} with 0 Operations Performed*.**

This matches the linear expectation of the basic operations for a pre-sorted array in the algorithm.

4.2. Algorithm Analysis

4.2.1. Algorithm Operations

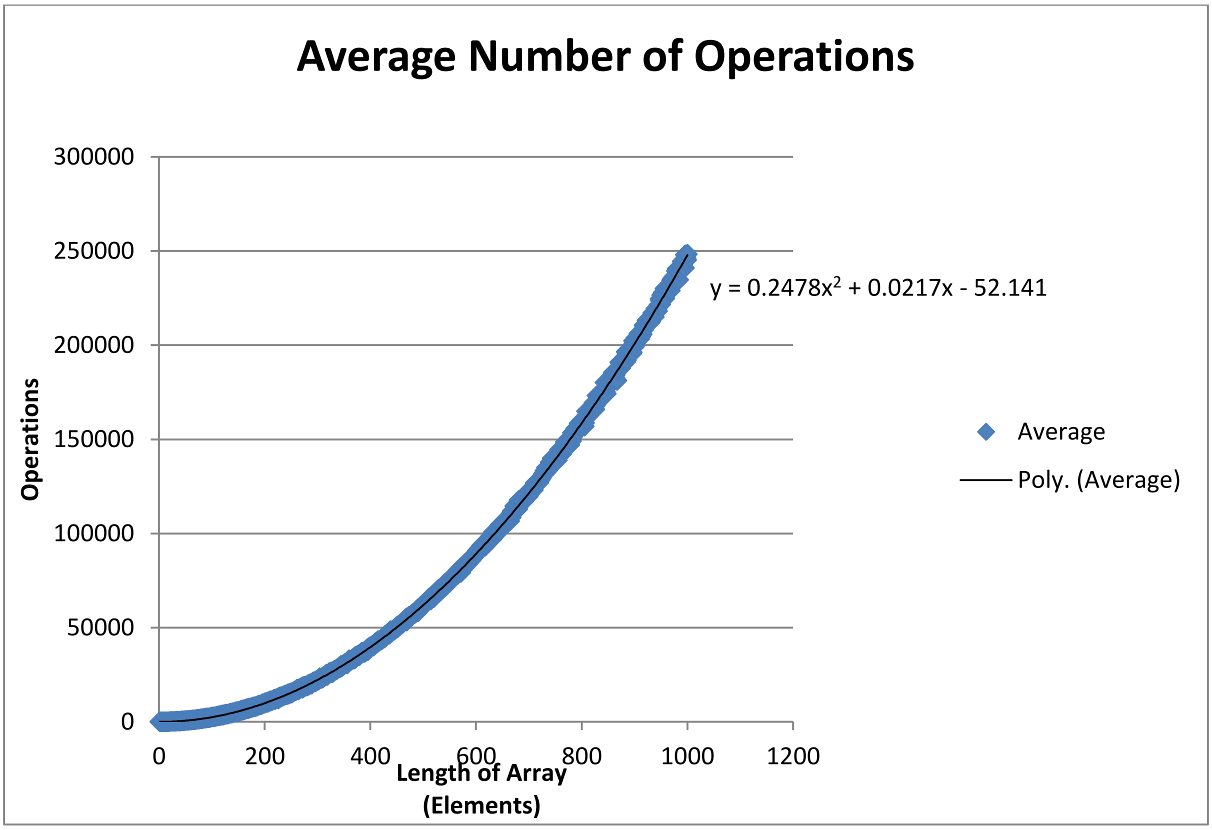
It was found that the comparison is performed with an average of n^2 / 4 times, where n is the number of elements. Based upon this assumption, the predicted operations for the algorithm were able to be determined and graphed below.



A trend implied by the formula and visible when graphed, is that as the length of the array increases, the number of operations is expected to increase as a quadratic.

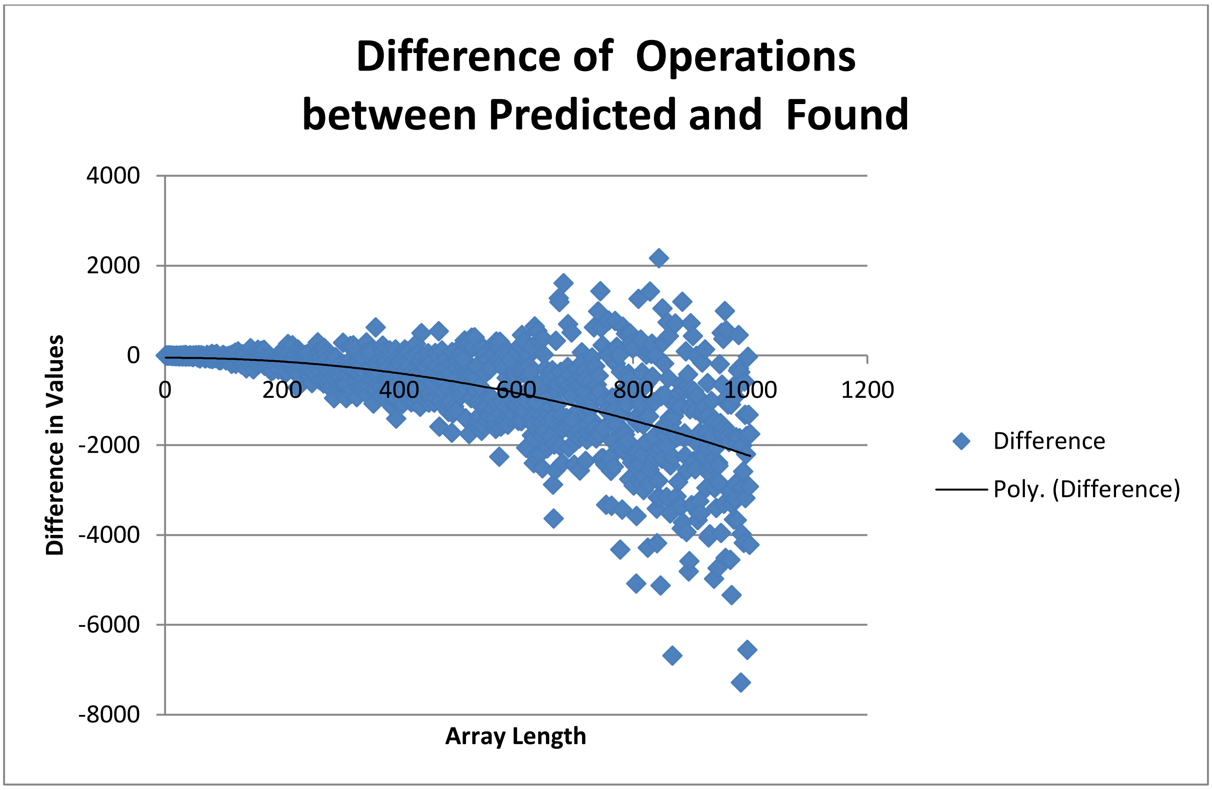
In order to determine the legitimacy of these predicted values, the algorithm was tested and graphed using the methods prescribed in Appendix 1.

The arrays used for testing, range in size from 1 to 1000 elements. These elements were randomly generated before being sorted in order to allow for the testing to be as unbiased as possible (For code generating random arrays, see Appendix 2). On completion of the testing, trends can be noted to have equal traits to that of the predicted operations.



Through the testing and subsequent graphing of the results, the operations rate of growth was confirmed as ***n^2*, *as a quadratic growth***. This average number of operations follows the order of growth as the previously established predicted trends.

This analysis of the tested values confirms the predicted rate of growth class as set by Levitin. Despite this trend, there are minor differences in the found values which have affected the number of operations and caused the rate of growth to vary slightly, although still within the same class.



It can be noted that the difference between the predicted number of operations and the found number through means of testing, does increase as the array length increases. However, as shown in the formula for the graphed trend-lines, the degree at which the difference increases do not have any bearing on the type of growth the algorithm was predicted to follow. This confirms that the predicted trend for the algorithm was accurately assessed and that the algorithm operations will drastically increase at a rate proportionate to the array size.

These differences in the number of operations which were unanticipated during the predicted case may be due to the functionality of the code. Within the code, the numbers assigned for sorting are randomly generated upon creation of the array, which may have generated some numbers more closely suited to the worst case, rather than the average.

Although there was no major impact on the operations, these differences may potentially have some bearing on the expected execution time.

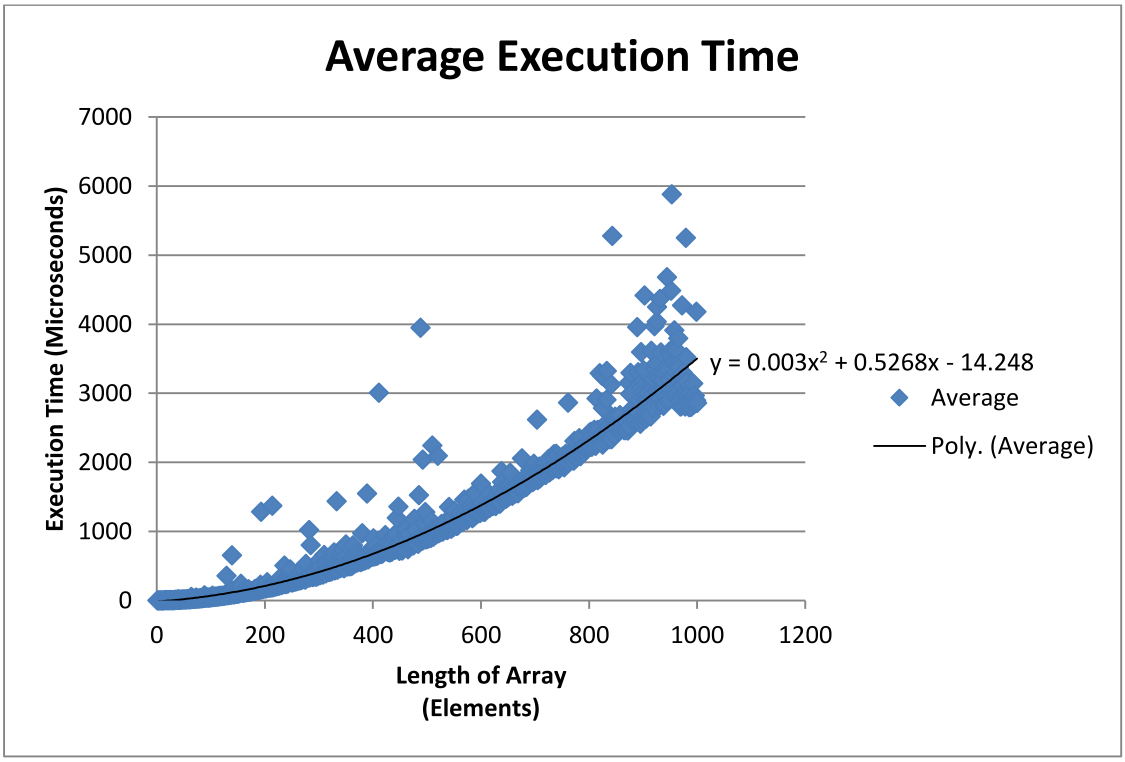
4.2.2. Algorithm Execution time

To assess the effectiveness of the algorithm, a practical application of the found operations was required. This would allow for the experiments to judge the efficiency for any unexpected overheads or optimisations when processing at run-time.

The same base code was used for consistency in data; however, the operation was changed so that the incremental counter of operations was no longer included. Instead of this, the time was recorded using the C++ *QueryPerformanceCounter* **(Microsoft, 2016)** method as shown in Appendix 5.

The method initialises a timer and a frequency counter that times the body of the code being run. The timer is closed once the body is run and this process is repeated for each element in the loop.

From this data, trends can be recognised regarding the runtime and overheads.



The above graph shows the average execution time of the experiment per array length in microseconds. The graphed trend can be seen to confirm the order of growth as a quadratic as the average execution time can be seen to increase proportionately to the increase in array size. This would indicate that the algorithm would not be suited to larger sized arrays.

The rate of growth for the execution times was potentially of a different rate to the operations value as previously stated, the number of operations increasing would change the average execution time similar to the manner shown in the above graph.

Overheads

Within the experimentation, it can be seen that there are inconsistencies that cause the data to sway from the expected growth rate as set by the number of operations. One trend of note is that while the line does begin to follow a quadratic rate of growth, it contains multiple outliers where the execution times exceed the trend patterns. These outliers would be unanticipated in a theoretical prediction as they are likely due to computer processing errors and are independent of the functionality of the code. This is likely the case as shown in the average times data table, which sees the different base values for the outliers which the average is calculated off, being significantly higher (upwards of 10 times the processing time) than the other the values for the same length of array. They become more prominent as the array sizes increase, which is potentially due to the extra processing required for arrays of greater sizes, however sheer increase in amount and indicates that these are outliers and not future trends of growth. See Appendix 6.

With the overheads present in the executed run time, there were no optimisations of the algorithm as the difference in operations trend can be seen to continue negatively.

References:

1. Levitin, A. (2007). *Introduction to the design and analysis of algorithms.* Boston, MA : Pearson.
2. Microsoft. (2016). *QueryPerformanceCounter Function*. Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx
3. Muhamma R. (2010). *Insertion Sort*. Kent University. Retrieved from http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/insertionSort.htm

Appendix 1. Code of Algorithm

The code the algorithm using the randomly generated arrays seen in Appendix 2.
In these experiments, this code runs the operations of the insertion algorithm for each array

```
for(int i = 1; i < Length; i++ ){
   value = numlist[i];
   tempvalue = i-1;
    while(tempvalue >= 0 && numlist[tempvalue] > value){
         numlist[tempvalue+1] = numlist[tempvalue];
         tempvalue--;
    }
    numlist[tempvalue+1] = value;

}
```

Appendix 2. Code for Generating Random Arrays

The code for generating the random value uses the srand function to seed a random value which then generate the values for each element in the array in a for loop.

```
srand(time(NULL));
for (int i = 1; i < 1001; i++){
int numlist[Length];

for(int i = 0; i < Length; i++ ){
 numlist[i]= rand() % 120;
}
….
….
  Length++;
```

Appendix 3. Code for Testing Correctness

During the test stage, the below code printed the array and the number of operations to the console.

```
void print(int numlist[]){
        for (int printcount = 0; printcount < Length; printcount++)
{
    cout << numlist[printcount] << ", ";
}
    cout << " With" << operationcount << "Operations Performed" << "\n";
}
```

The code for testing the correctness of the algorithm is below. It is the same as the regular algorithm, but does not randomly generate the values to be tested. Instead the values are taken from the test array specified on line 37 and 38.

```
34     int test1[10];
35     int test2[10];
36
37     test1[] ={1,3,5,7,9,2,4,6,8};
38     test2[] ={1,2,3,4,5,6,7,8,9};
....
....
41     for(int i = 1; i < Length; i++ ){
42         value = test[i];
43         tempvalue = i-1;
44          while(tempvalue >= 0 && test[tempvalue] > value){
45                test[tempvalue+1] = test[tempvalue];
46                tempvalue--;
47          }
48          test[tempvalue+1] = value;
49
50     }
....
....   print();
```

Appendix 4. Code for Counting Basic Operations

The code for counting number of basic operations is an incremental counter that adds for every time the operation is run as shown below.

```
While (tempvalue >= 0 && numlist[tempvalue] > value){
operationcount++;
```

Appendix 5. Code for Measuring Runtime

The code for measuring the run time of the algorithm uses a Microsoft native counter called *QueryPerformanceCounter.* This counts the number of ticks between two performance points and calculates the run time in microseconds by dividing this length by the frequency of ticks

```
        QueryPerformanceFrequency(&frequency);
        QueryPerformanceCounter(&timerstart);
                Body of code to be timed
        QueryPerformanceCounter(&timerend);
        …
        …
        elapsedTime = (timerend.QuadPart - timerstart.QuadPart) * 1000000.0 /
frequency.QuadPart;
```

Appendix 6. Outlier Data

This table shows a section of the outlier data collected between arrays the size of 915-1000 elements. The boxes shaded in red are the outliers which are more than the average.

Array Size	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Average
915	6309.37	2384.77	3096.91	2568.34	2891.75	3380.9	2894.99	6786.65	3455.41	2384.77	3615.386
929	6034.55	2506.79	2425.81	2548.37	3039.68	2917.66	4489.88	3313.42	2397.73	2424.19	3209.808
952	14896.1	2524.07	2614.78	2869.61	4269.59	6336.36	3130.39	3029.96	2601.28	2568.34	4484.048
953	2806.98	2618.01	2546.75	3356.07	2526.23	3095.83	7200.21	2507.87	3219.47	28900.2	5877.762
979	2735.17	2738.41	3387.38	2767.03	3505.17	2819.94	3396.56	4178.3	2656.35	24305.6	5248.991
999	2974.35	13262.3	2837.22	2997.03	2884.73	2848.02	3105.01	2943.58	2722.22	5216.59	4179.105