Programming Assignment 01

1DV516

Name: Tadj Cazaubon

Student: tc222gf

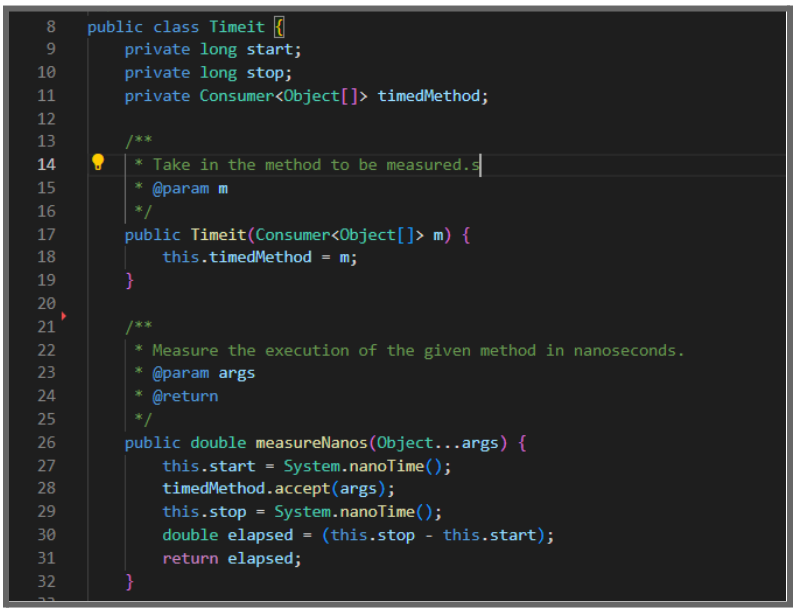
Email: tc222gf@gmail.lnu.se

Prerequisites

I ran all readings on the same Windows desktop system. I fixed the CPU clock speed to 4.4GHz and closed all background processes besides those needed by the OS.

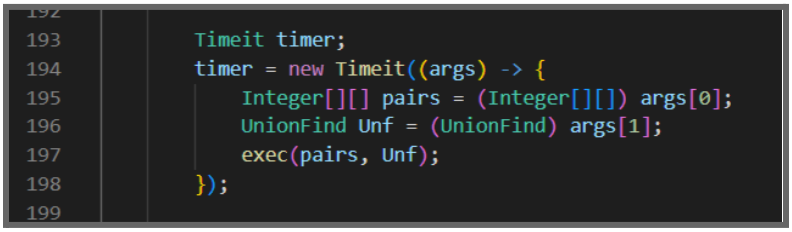
The Timeit Class

To time the execution of methods as specified in Problem 3, I've created the Timeit class. This works similarly to the python package of the same name, where a code snippet/ method can be run, and a nanosecond timer of its execution is given. Within my Timeit implementation a Consumer object can be passed in along with any number of args during instantiation. The method can then be run with the passed args. The only downside of this is the need for arguments passed to be an object, meaning no primitive data types can be passed without being wrapped within some kind of object.



In practice, args must be specified unless specifically programmed for, meaning I specify a simple anonymous function to separate the arguments and pass them to the method being timed correctly. I figure that the impact of this extra step on the final times is small, and when looking at the growth of an algorithm on a millisecond scale when the initial separation step takes nanoseconds, the results should remain consistent.

An example of one of these is the one used for the UnionFind analysis in Problem 4:



Here, the exec method is the thing we are looking for. This method, as we will see later, takes multiple milliseconds in some cases.

The Race Package

Each Set of methods for getting/graphing data for a given Problem number is within the race package. Here is where you'll find the definitions for the timers and graphs created during testing and analysis.

1. Problem 04

As specified in Problems 1 and 2, I implemented a simple UnionFind (Quick Find), along with both a Quick Union and Weighted Union Find with path compression. While it was only asked to make the QuickFind and one faster UnionFind, I figured that due to its simplicity, I would just do QuickUnion in addition to my choice of Weighted Union Find with path compression. To make things such as timing their execution, I use a parent "UnionFind" class which the others then inherit. I tested the implementations against one another at multiple fixed sizes in a manner similar to the lecture slides.

1.1. Procedure:

A graph object is created for every individual set of readings, specifying the graph details. These are also passed down to the individual graphs for closer inspection. The "run" methods created specify the fixed array sizes, steps of the readings, and the start of the readings as to control the number of points and area of observation.

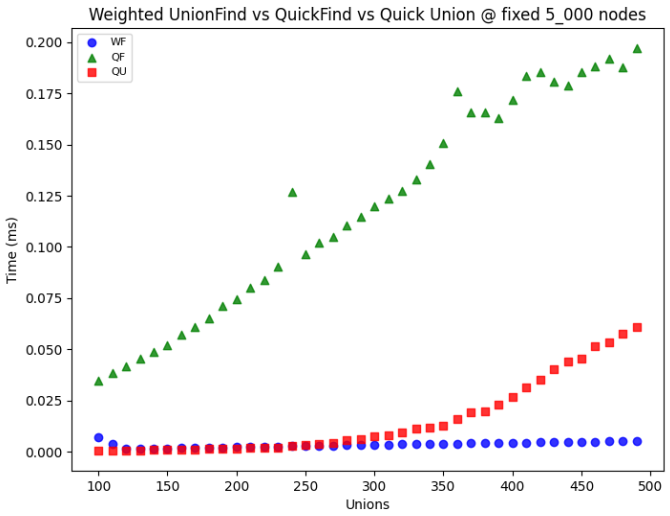
```
35
36          plt = new Plotter<>(path:"uf/WUFvsQF_10_000_max.png",
37                              x_label:"Unions", y_label:"Time (ms)",
38                                      Plotter.Type.LINEAR,
39                              title:"Weighted UnionFind vs Quick Find  @ fixed 10_000 nodes");
40
41          race.runWFvQF(SIZE:10000, STEPS:100, START:1000, plt);
42
43
```

These points are used to generate the x-values as a list which are passed on the other methods. For each x-value (Union Size) N, an N-size random array of Integer tuples is generated to serve as the integer pairs for unions. A simple for-loop of the points and calling the union method is done, and the time saved to a list of samples. The number of samples is predefined at 100 for all the experiments. These are then given to the sampleMean method I created in the Util class. This simply finds the upper and lower bound of the readings via the IQR to enable filtering of outliers. This way I can get rid of reading which may have been affected by large changes in my computer's state or other irregularities. After this it returns the mean with the outliers excluded.

```
37          */
38      public static double sampleMean(double[] samples) {
39          Arrays.sort(samples);
40          double q1 = Quartile(samples, percentile:25);
41          double q3 = Quartile(samples, percentile:75);
42          double iqr = q3 - q1;
43
44          double ub =  q3 + 1.5*iqr;
45          double lb =  q1 - 1.5*iqr;
46
47          int count = 0;
48          double sum = 0;
49
50          for (int i = 0; i < samples.length; i++) {
51              double value = samples[i];
52              if (value >= lb && value <= ub) {
53                  sum += value;
54                  count++;
55              }
56          }
```

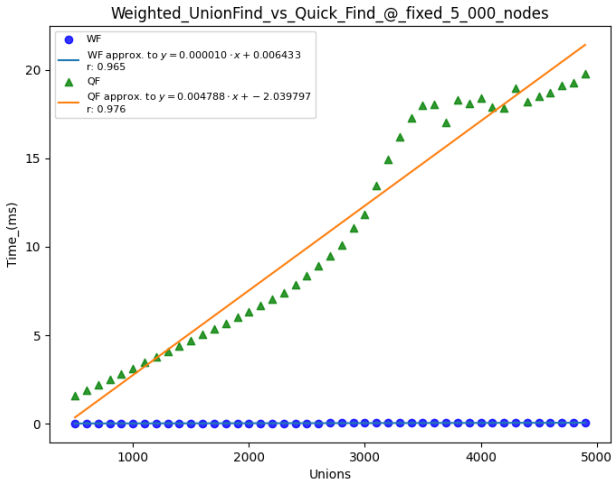
1.2. Overview

Each implementation is plotted both on a comparison graph, and an individual graph with the same coordinates to allow for closer inspection.



Here we see the 3 implementations graphed for a fixed 500 array size of random integers. QuickFind and Weighted UnionFind as expected, seem nearly linear, whilst QuickUnion seems linear at first before having the characteristic sharp increase in time near the 50% mark. This seemingly is due to the increase in the depth of the tree after several sequential unions with no consideration for the subtrees during unions, meaning each find operation during unions therefore takes longer each time. This then is not seen in the Weighted UnionFind with path compression implementation. This is because the height of the trees is considered with each union and longer subtrees are further reduced with each find. This means that while the depth of the tree must of course increase, it increases at a far slower rate (Staying within log N with N being the number of elements in the disjoint set).

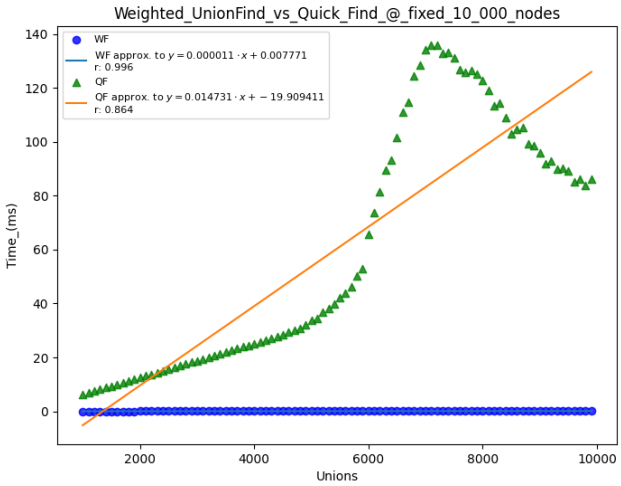
I will mainly compare the WeightedUnionFind with Path compression to the QuickFind implementation. Firstly, a look at the shape of the graphs at different sizes. At 5000 elements, the difference between the growth of these implementations are worlds apart.

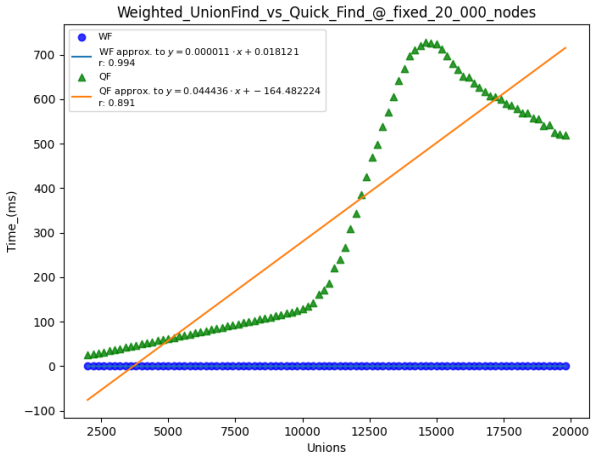


The QuickFind seems to follow a fairly linear growth pattern, while the Weighted approach seems nearly constant in comparison. In the next section we will see that it does in fact increase. This is almost exactly as I would have expected. However, this sharp increase and seeming decline is unexpected around the 60% mark. I would be drawn to the same reasoning as used with the QuickUnion, that the lengths of the trees simply spike up due to successive union operations. This doesn't however explain the seeming decline in time increase right after the spike. This only becomes more confusing for larger numbers of elements.

Looking now at the same implementations with 10,000 random elements, the spike and subsequent decrease is even more exaggerated.

Note: The line of best fit calculated here is an artefact of the state of the graphing script I created before the more recent change. This can be ignored in this case.





This seems, however, to somewhat decrease in the case of even larger fixed-size element amounts. As we can see in the case of 20,000 elements, this spike and subsequent drop is not as pronounced visually, but still present.

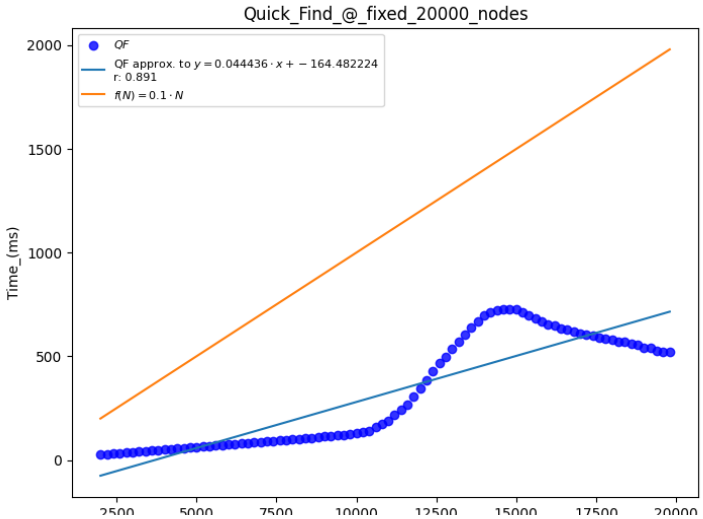
1.3. Analysing Growth

1.3.1. QuickFind

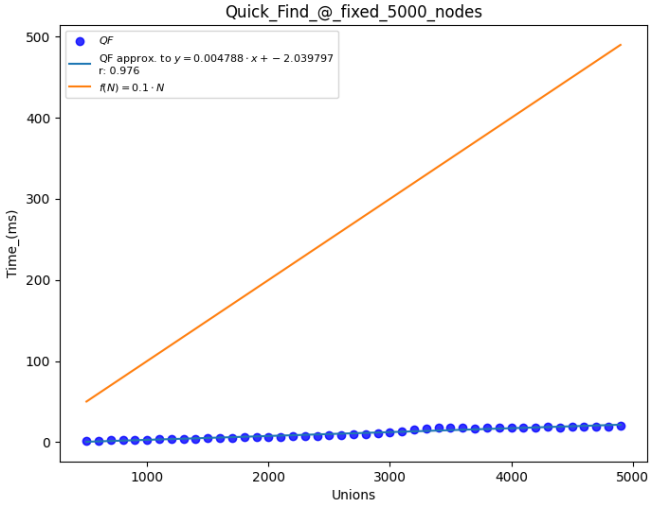
To properly categorise the growth of these implementations within the bounds we have looked at, we use Big Theta, Big Oh, and Big Omega.

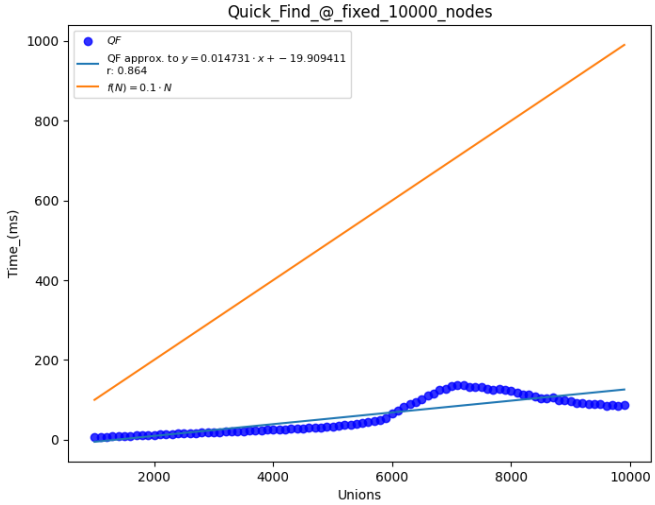
Firstly, the analysis of QuickFind. We look at the example with the largest number of elements. We're of course looking to see if it follows a linear time O(N), meaning there must be some positive constants c and n_0 such that for all points T(N) on our QuickFind graph, $T(N) \leq c \cdot N, \forall N \geq n_0$. Looking at the graph of just the QuickFind times, I can come up with the function line $g(N) = 0.1 \cdot N$, seen below as a possible upper bound.

We can see that for all measured values of the QuickFind function T(N), g(N) is greater. This means that $T(N) < 0.1 \cdot N, \ \forall \ N > 2000$, as that's the smallest N measured. This means we could say c = 0.1, n_0 = 2000. This would therefore mean that the QuickFind implementation for this range of values is O(N).

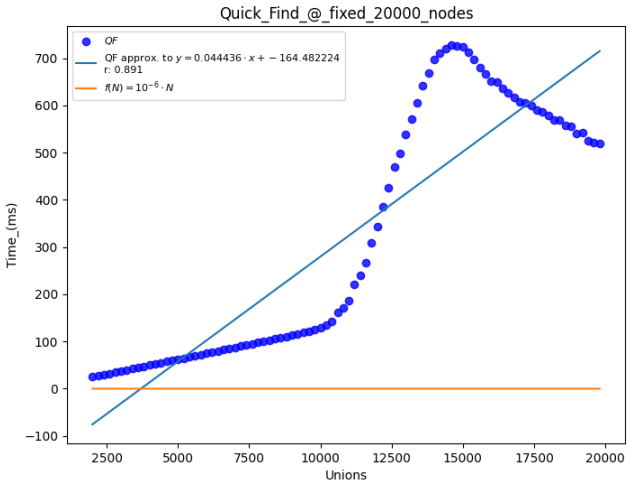


This is of course also true using the same function for the runs of the implementation with a smaller fixed size as seen below. While c will remain the same here, the minimum value n will be 1000 and 500 respectively.





We now want to see if it follows Ω(N). Looking once again at the graphed data, I come up with the equation $g(N) = 10^{-6} \cdot N$. We look for positive constants c and n_0 such that for all points on the QuickFind time graph T(N), $T(N) \geq c \cdot N, \ \forall \ N \geq n_0$. Here we see that the chosen $g(N) = 10^{-6}$ fulfils this need for our run with 20,000 elements.

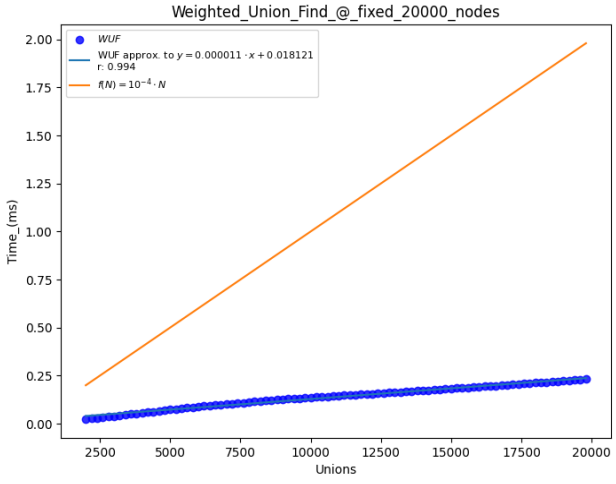


We can see for all measured values of the QuickFind function T(N), T(N) > g(N). This means that $T(N) > 10^{-6} \cdot N, \ \forall \ N > 2000$ as that's the smallest N measured. This means we could say $c=10^{-6}$, f(N) = N, $n_0 = 2000$. This would therefore mean that the QuickFind implementation for this range of values is Ω(N).

As I've now established the QuickFind implementation I've created is both O(N) and Ω(N) for the range of values I've created, I can say that this implementation is Θ(N) for the range of values tested.

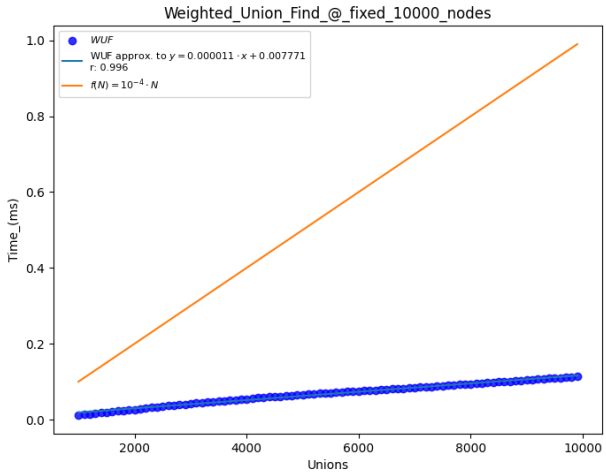
1.3.2. Weighted Union Find

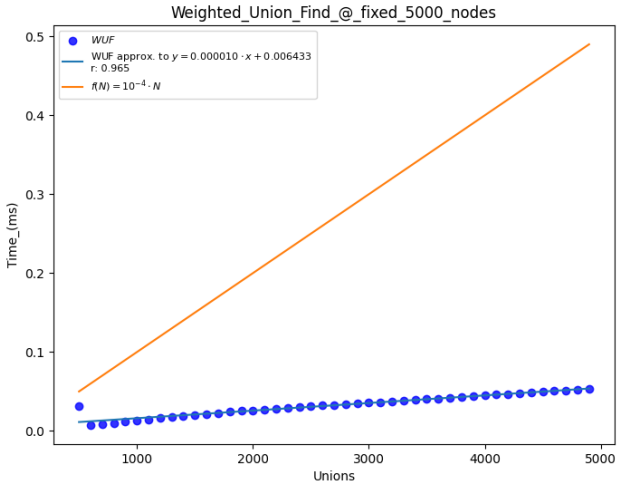
We look at the example with the largest number of elements. We're looking to see if it follows a linear time O(N), meaning there must be some positive constants c and such that for all points T(N) on our WeightedUnionFind graph, $T(N) \leq c \cdot N, \ \forall \ N \geq n_0$. Looking at the graph of just the WeightedUnionFind times, I can come up with the function line $g(N) = 10^{-4} \cdot N$, seen below as a possible upper bound.



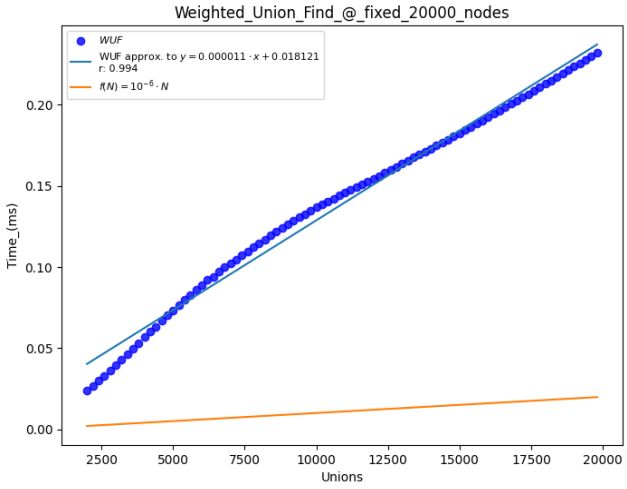
We can see that for all measured values of The Weighted function T(N), g(N) is greater. This means that $T(N) < 10^{-4} \cdot N, \ \forall \ N > 2000$ as that's the smallest N measured. This means I could say $c = 10^{-4}$, f(N) = N, $n_0 = 2000$. This would therefore mean that the Weighted implementation for this range of values is O(N).

This function then is of course also valid as an upper bound for the smaller fixed size runs tested, as seen below. While c remains constant, the minimum value n_0, will be 1000 and 500 respectively.



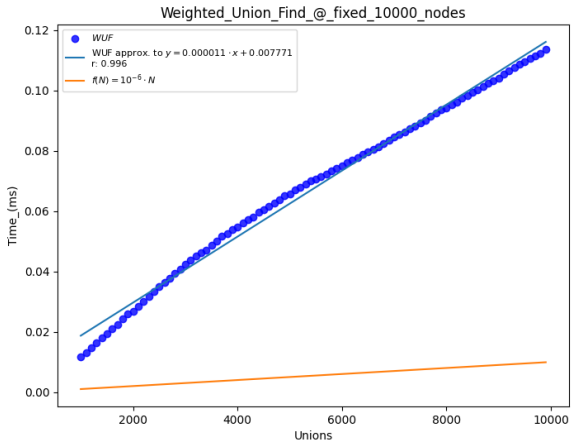


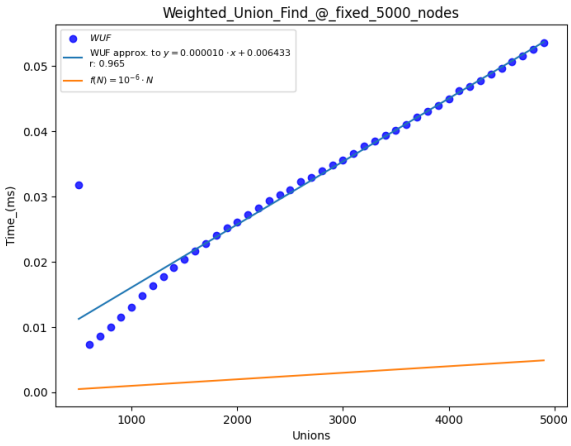
We now want to see if it follows Ω(N). Looking once again at the graphed data, I come up with the equation $g(N) = 10^{-6} \cdot N$. We look for positive constants c and n_0 such that for all points on the Weighted Union Find time graph T(N), $T(N) \geq c \cdot f(N)$, $N \geq n_0$. Here we see that the chosen g(N) fulfils this need for our run with 20,000 elements.



We can see for all values of the Weighted function, $T(N) > 10^{-6} \cdot N$, $\forall$ $N > 2000$ as that's the smallest N measured. This means I could say $c = 10^{-6}$, $f(N) = N$, $n_0 = 2000$. This would therefore mean that the Weighted Union Find implementation for this range of values is Ω(N).

This function then is of course also valid as an upper bound for the smaller fixed size runs tested, as seen below. While c remains constant, the minimum value n_0, will be 1000 and 500 respectively.





As I've now established the WeightedUnionFind implementation I've created is both O(N) and Ω(N) for the range of values I've created, I can say that this implementation is Θ(N) for the range of values tested.

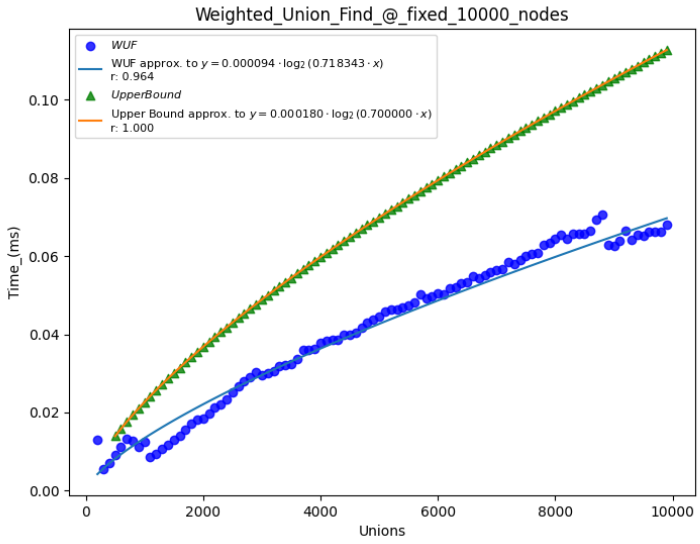
It may have been more accurate to also test for logarithmic time complexity in hindsight to be more accurate, however, by definition, both implementations are ΘN for the values tested.

These results are about as I would expect. The Weighted Union Find is by far the slowest growing algorithm in terms of the time taken for its union operation. However, a closer look at the complexity of the Weighted implementation leads to the realisation that it may be better described by Logarithmic time complexity, though it can be said to be Θ(N). With each operation taking the height of the subtrees into account, the maximum height of the trees can be shown to never exceed log(N), as well as being continually flattened via path compression.

1.3.2.1 Weighted Union Find - Logarithmic

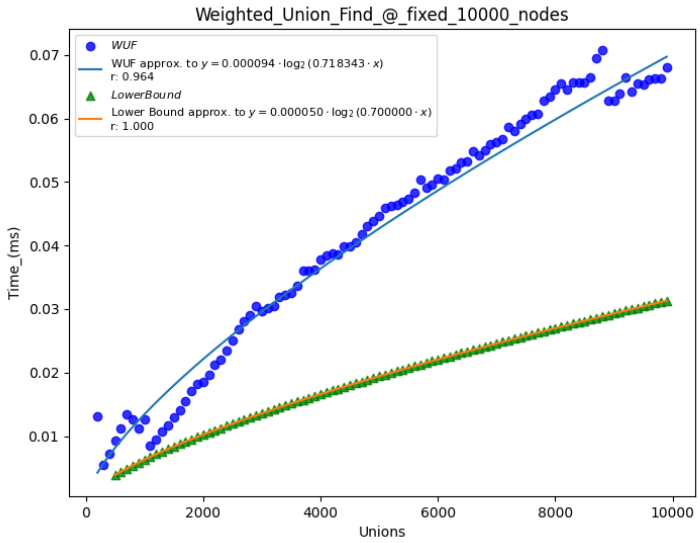
Within the Weighted Union Find implementation, the time complexity of the union operation is dominated by the find operation. The time complexity of the find operation in turn is dictated by the heighted on the tree. The weighted union optimization ensures that the depth of the tree representing the disjoint sets is minimised. When performing a union operation between two subsets, the smaller subset is attached to the larger one. This guarantees that the depth of the resulting tree is at most log(N), where N is the total number of elements. Path compression then helps in flattening the tree during find operations, which in turn helps maintain the logarithmic depth of the trees, contributing to the overall efficiency of the union operation. We will now test validity of O(log n)

We're looking to see if it follows a logarithmic time O(log N), meaning there must be some positive constants c and n_0 such that for all points T(N) on our graph, $T(N) \leq c \cdot g(N), \ \forall \ N \geq n_0$. g(N) is some function g(N) = $log(a \cdot N)$, where a is some constant, therefore evaluating to O(log N). Looking at the graph of just the WeightedUnionFind times, I can come up with the function line $g(N) = \log(0.7 \cdot N)$, $c = 1.8 \cdot 10^{-4}$, seen below as a possible upper bound.



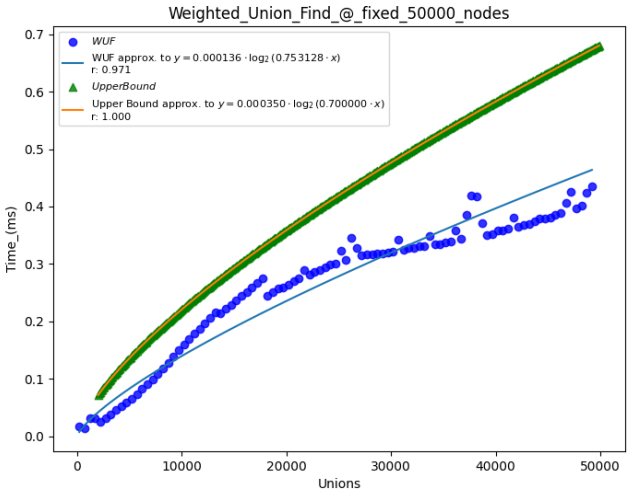
We can see that for all measured values of the function T(N), $1.8 \cdot 10^{-4} \cdot g(N)$ is greater. This means that $T(N) \leq 1.8 \cdot 10^{-4} \cdot \log(0.7 \cdot N), \ \forall \ N > 500$ as that's the smallest N measured. This means I could say $c = 1.8 \cdot 10^{-4}$, $g(N) = \log(0.7 \cdot N)$, $n_0 = 500$. This would therefore mean that the Weighted implementation for this range of values is O(log(0.7 · N)). This would of course evaluate to O(log(0.7) + log(N)), which may have log(0.7) be ignored as a constant and used as O(log(N)).

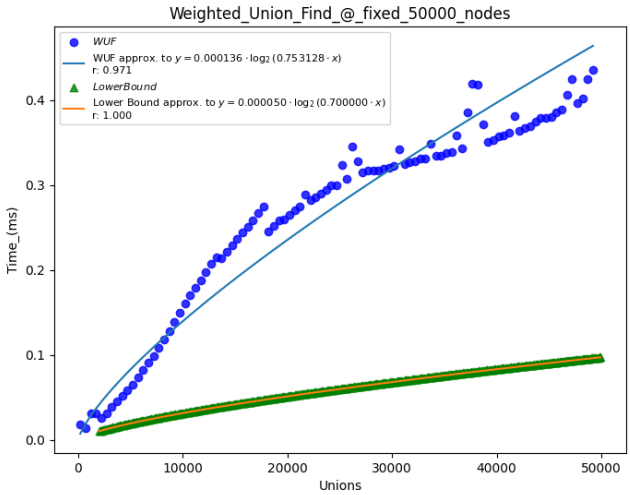
We're looking to see now if it follows a logarithmic time Ω(log N), meaning there must be some positive constants c and n_0 such that for all points T(N) on our graph, $T(N) > c \cdot g(N), \ \forall \ N \geq n_0$. g(N) is some function g(N) = $log(a \cdot N)$, where a is some constant, therefore evaluating to O(log N). Looking at the graph of just the WeightedUnionFind times, I can come up with the function line $g(N) = \log(0.7 \cdot N)$, $c = 0.5 \cdot 10^{-4}$, seen below as a possible lower bound.



We can see that for all measured values of the function T(N), $0.5 \cdot 10^{-4} \cdot g(N)$ is lesser. This means that $T(N) > 0.5 \cdot 10^{-4} \cdot \log(0.7 \cdot N), \ \forall \ N > 500$ as that's the smallest N measured. This means I could say $c = 0.5 \cdot 10^{-4}$, $g(N) = \log(0.7 \cdot N)$, $n_0 = 500$. This would therefore mean that the Weighted implementation for this range of values is Ω(log(0.7 · N)). This would of course evaluate to Ω(log(0.7) + log(N)), which may have log(0.7) be ignored as a constant and used as Ω(log(N)).

We can also see this hold true for a larger number of Nodes in the disjoint set, up to 50,000. Below I have tested with an upper bound of $y = 3.5 \cdot 10^{-4} \cdot \log(0.70 \cdot N)$ and lower bound of $y = 0.5 \cdot 10^{-4} \cdot \log(0.70 \cdot N)$ respectively, both with an initial value of 2000.





As I've now established the WeightedUnionFind implementation I've created is both O(log(N)) and Ω(log(N)) for the range of values I've created, I can say that this implementation is Θ(og(N)) for the range of values tested.

2. Problem 07

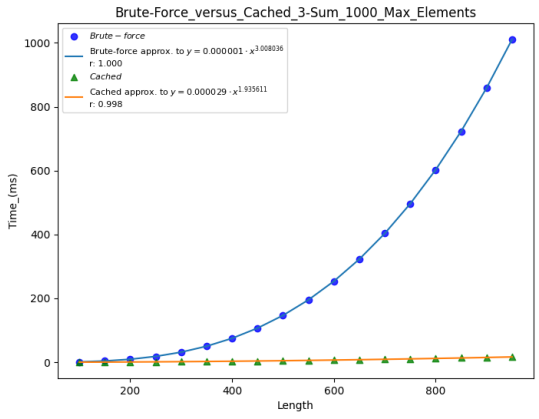
2.1. Procedure:

Separation of responsibility is almost identical to the outline of the previous section. To make everything simpler, I've done two things. Firstly, is using a normal Twosum implementation within the threesum implementations, though it is only used in the cached threesum. Second, is structuring the cached threesum implementation as a child class of the brute force threesum implementation.

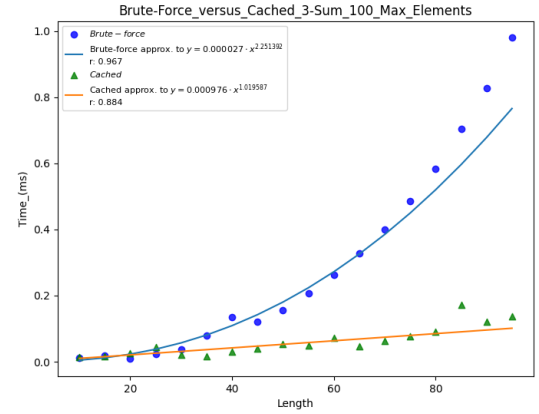
The target sums for each Threesum array size are predetermined at the beginning of the run and are the same between both implementations, though the arrays are randomly generated for each size. Timing, sampling and measurement are nearly identical to the previous section.

2.2. Overview

Each implementation is plotted both on a comparison graph, and an individual graph with the same coordinates to allow for closer inspection. While analysis would have been more comprehensive if I could have used larger array sizes, the time taken would be far too great for the brute-force implementation. The growth of these is almost exactly as I would expect, with their approximate function equivalents approaching being cubic and quadratic respectively.

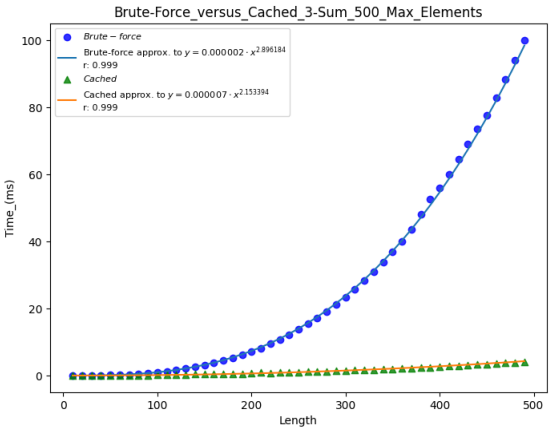


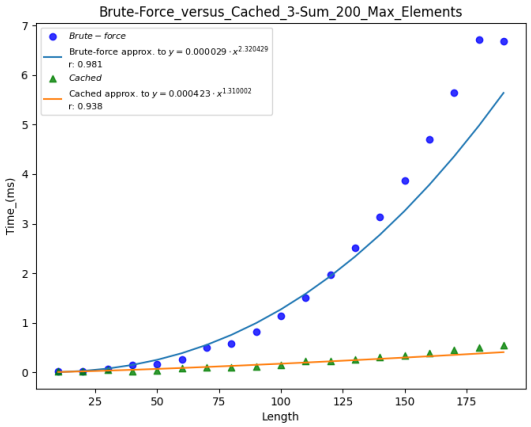
Here we see the brute-force and cached threesum approaches up to a maximum size of 1000 elements. As expected, the difference in times taken is huge, with the growth of the cached implementation seeming almost constant in comparison to the brute-force method. Each was approximated to a power-law via linear regression done on their log-log plot equivalents. The brute-force approach has a slope which is almost perfectly cubic, and the cached approach's slope is almost perfectly quadratic.



Beginning with a run which stopped after the 100-element point, we can see that while the difference in their growth is immediate, their slopes have yet to sufficiently approach their true time complexity equivalent.

After this, we can see a gradual conformity to the expected respective time complexities. This can be demonstrated nicely via the change seen between a run up to 200 max elements and a subsequent run up to 500 elements (right and left images respectively). The approximate slope of each run becomes much closer to what we would expect.





2.3. Analysing growth

2.3.1. Brute-Force Threesum

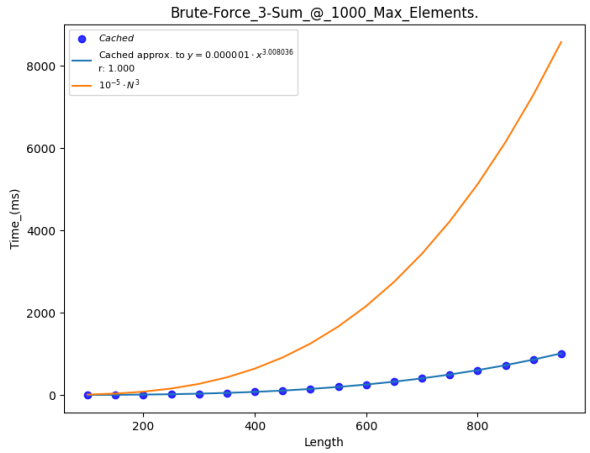
I was asked in P5 to “Implement the $O(N^3)$ brute-force variant of 3sum”. I did this by implementing the following class.

```java
public class ThreeSum {
    TwoSum two;
    public ThreeSum () {
        two = new TwoSum();
    }

    public List<int[]> threeSum(Integer [] nums, Integer target) {
        List<int[]> result = new ArrayList<>();
        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < nums.length; j++) {
                for (int k = 0; k < nums.length; k++) {
                    if (i==j || i == k || j== k) {
                        continue;
                    }

                    if (nums[i] + nums [j] + nums [k] == 0) {
                        result.add(new int[] {nums[i], nums[j], nums[k]});
                    }
                }
            }
        }
        return result;
    }
}
```

This implementation, taken partly from the lectures, must at worst loop for N^3, though only access the array for *roughly* $N^3 - 2N^2 - N + 2N$. This for large values of N, will approach N^3 in time.

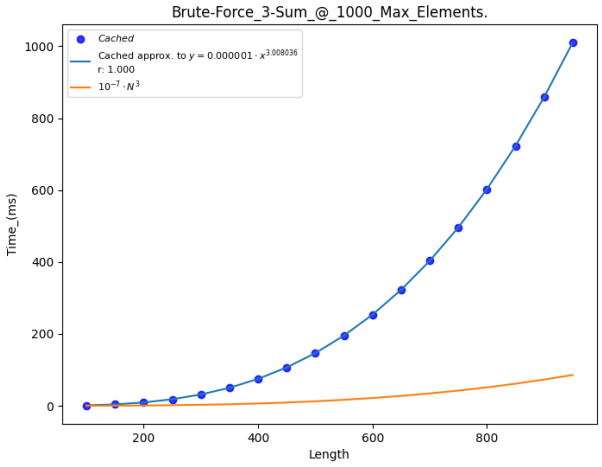
To properly categorise the growth of these implementations within the bounds we have looked at, we use Big Theta, Big Oh, and Big Omega.

We look at the example with the largest number of elements. We're looking to see if it follows cubic time $O(N^3)$, meaning there must be some positive constants c and n_0 such that for all points on our graph T(N), $T(N) \leq c \cdot N^3, \ \forall \ N \geq n_0$. Looking at the graph of just the Threesum times, I can come up with the function $g(N) = 10^{-5} \cdot N^3$, seen below as a possible upper bound.



While it seems that they begin at similar readings, the Threesum begins at 1.17 ms, while the upper bound function begins at 10 ms. This means $c = 10^{-5}$, $n_0 = 100$, since that's the smallest measurement taken. We can say $T(N) < c \cdot N^3, \ \forall \ N > n_0$. This would therefore mean that T(N) can be said to be $O(N^3)$.

I'll skip the graphed runs with smaller maximums, as these are essentially slices of the larger graph above.



To see if T(N) can be described as Ω)N), we must find some positive constants c and n_0 such that for all points on our graph T(N), $T(N) \geq c \cdot N^3, \ \forall \ N \geq n_0$. Looking at the same graph from above, especially given the approximate function, we can come up with the function $g(N) = 10^{-7} \cdot N$, seen here as a possible lower bound.

While it may seem at first glance that these begin at the same point, the Threesum begins at 1.17 ms, while the lower bound function begins at 0.1 ms. This means $c = 10^{-7}$, $n_0 = 100$, since that's the smallest measurement taken. We can say that $T(N) > c \cdot N^3, \ \forall \ N > n_0$. This would therefore mean that T(N) can be said to be $\Omega(N^3)$.

Note: The brute-force Threesum graphs have been incorrectly labelled as "Cached". The data points however, are from the brute-force implementation.

Knowing that the brute-force Threesum is both $O(N^3)$ and $\Omega(N^3)$, we can say then that it is $\Theta(N^3)$. This is exactly as I would expect, and as I had laid out in my prior explanations.

2.3.2. Cached Threesum

I was asked in P6 to “a version of 3sum with an upper bound that is lower than $O(N^3)$”. I did this by implementing the following class.

```java
public class ThreeSumCached extends ThreeSum{
    public ThreeSumCached() {
        super();
    }
    public List<int[]> threeSum(Integer [] nums, Integer target) {
        List<int[]> result = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            int current = nums[i];
            List<int[]> twoSumRes = this.two.twoSum(nums, target-current, i+1);

            if(twoSumRes.isEmpty()) continue;

            for (int[] pair : twoSumRes) {
                int[] trip = {current, pair[0], pair[1]};
                result.add(trip);
            }
        }
        return result;
    }
}
```

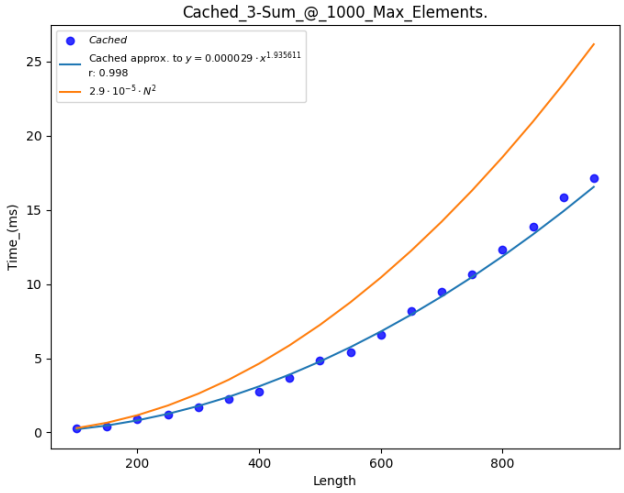
This implementation of Threesum uses the cached implementation of TwoSum below to find all valid pairs of values for each value in the number array. The TwoSum implementation operates at O(N) as it iterates over the array segment given, adds each to a hashmap, and refers to that when looking for value pairs. The value pairs are then iterated over and made into triplets which are later returned. This nested iteration could be considered though this is variable and small in comparison to the overall iteration of the number array.

```java
/**
 * 2-Sum implementation with caching to allow for linear rather than
 * Quadratic growth.
 */
public class TwoSum {

    public List<int[]> twoSum (Integer[] nums, Integer target, Integer start) {
        List<int[]> result = new ArrayList<>();
        Map<Integer, Integer> seen = new HashMap<>();
        for (int i = start; i < nums.length; i++) {
            int complement = target - nums[i];
            if (seen.containsKey(complement)) {
                int[] pair = {nums[seen.get(complement)], nums[i]};
                result.add(pair);
            }
            seen.put(nums[i], i);
        }

        return result;
    }
}
```

For each iteration in the number array, a call to the TwoSum method from that index onward in the array is made. This means that we don’t iterate over the entire array each time. The time complexity is better than O(N) for each element in the ThreeSum method. It's O(N) for the first element, then O(N−1) for the second element, O(N−2) for the third element, and so on. The time complexity therefore is a sum of an arithmetic progression, the sum of the first N natural numbers. $N+N-1+N-2+\ldots+1$ which can be expressed as $N \cdot (N+1)/2$. This would therefore simplify to N^2. We can therefore expect our results to show that the cached Threesum implementation performs as $O(N^2)$.

To properly categorise the growth of these implementations within the bounds we have looked at, we use Big Theta, Big Oh, and Big Omega.

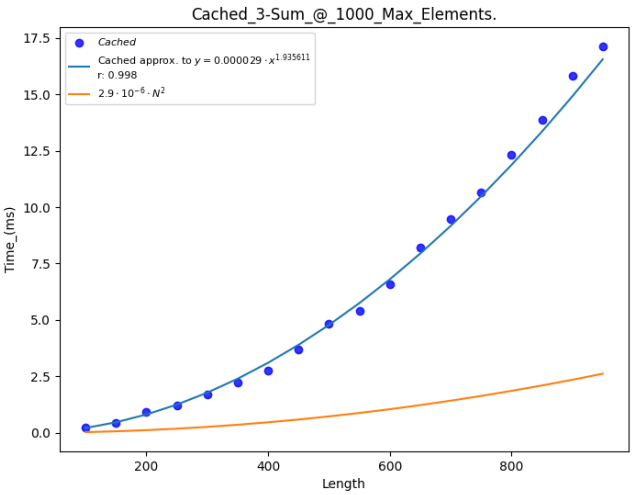
We look at the example with the largest number of elements. We’re looking to see if it follows quadratic time $O(N^2)$, meaning there must be some positive constants c and n_0 that for all points on our graph, T(N), $T(N) \leq c \cdot N^2, \forall N \geq n_0$. Looking at the graph of just the Threesum times, informed by the approximated exponential function, I can come up with the function $g(N) = 2.9 \cdot 10^{-5} \cdot N^2$, seen below as a possible upper bound.



Though they appear to begin from similar points, the measured points begin at 0.245 ms, while the upper bound function begins at 0.290 ms. Given all points on the upper bound function are greater than the measured points, $c = 2.9 \cdot 10^{-5}$, $n_0 = 100$ as that's the smallest array size measured. We can therefore say $T(N) < c \cdot N^2, \forall N \geq n_0$. With this, we can say that the implementation of the cached Threesum here is $O(N^2)$.

We skip the analysis of runs with smaller maximum array sizes, as these are essentially subgraphs of the one already discussed above.

To see if the implementation above is $\Omega(N^2)$, we look to see if it follows quadratic time $\Omega(N^2)$ meaning there must be positive constants c and n_0 that for all points on our graph T(N), $T(N) \geq c \cdot N^2, \forall N \geq n_0$. Looking at the graph of just the Threesum times, and informed by the approximated exponential function, I can come up with the function $2.9 \cdot 10^{-6}$, seen below as a possible lower bound.



Though they appear to begin from similar points, the measured points begin at 0.245 ms, while the lower bound function begins at 0.029 ms. Given all points on the lower bound function are lesser than the measured points, $c = 2.9 \cdot 10^{-6}$, $n_0 = 100$ as that's the smallest array size measured. We can therefore say $T(N) > c \cdot N^2, \forall N \geq n_0$. With this, we can say that the implementation here is $\Omega(N^2)$.

Knowing that the cached Threesum is both $O(N^2)$ and $\Omega(N^2)$, we can say then that it is $\Theta(N^2)$. This is exactly as I would expect, and as I had laid out in my prior explanations.

3. Problem 08

3.1. Overview

```java
public static double mean(double[] samples)   {
    double sum = 0;
    for(double i: samples) sum+=i;
    return sum/samples.length;
}

public static double stdDev(double[] samples) {
    int n = samples.length;
    double sum = 0.0;
    double mean = mean(samples);

    for (double num: samples) sum += Math.pow(num-mean, 2);

    return Math.sqrt(sum/(n-1));
}
```

To calculate the mean and standard deviation of my samples, I added the following methods to the Util static class I defined earlier. It holds a number of “utility” methods.

```java
public static void writeCSV(String path, String[] headers, String[][] content) {
    try (FileWriter writer = new FileWriter(path)) {

        String joined = String.join("\t", headers);
        writer.append(joined);
        writer.append('\n');

        for (int row =1; row<content.length;row++) {
            joined = String.join("\t", content[row]);
            writer.append(joined);
            writer.append('\n');
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

A writeCSV method was added to the static Util class so we can write the results of the findings to a file representable in a tabular format.

```java
public void open(int row, int col) {
    assertValid(row, col);

    if (isOpen(row, col)) return;

    grid[row-1][col-1] = true;
    openCount+=1;


    /**
     * Now we connect the index to adjacent sites.
     * If in the top row, connect to the top node.
     * If in the bottom row, connect to the bottom node.
     */
    int index = DDtoD(row, col);
    if (row == 1) wuf.union(index, top);
    if (row == size) wuf.union(index, bottom);

    /**
     * Put the offsets for the four adjacent rows
     */
    int[][] offsets = { {-1, 0},
                        { 1, 0},
                        { 0,-1},
                        { 0, 1} };

    for(int[] off: offsets) {
        int nRow = row + off[0];
        int nCol = col + off[1];

        /**
         * If the adjacent site is open and valid, connect them.
         */
        if (isValid(nRow, nCol) && grid[nRow-1][nCol-1]) {
            int nIndex = DDtoD(nRow, nCol);
            wuf.union(nIndex, index);
        }
    }
}
```

To properly simulate percolation of an N x N grid, I created the Percolation class, with the needed methods for things such as opening a site, checking a site state, getting the number of open sites, etc.

The open method in particular is important, with a large effort made to simplify it. Firstly if either the coordinates are invalid, or the cell is already open, we throw an error or return respectively. We open the cell and increment the count. We connect the cell to its adjacent cells (up, down, left, right). Cells are connected by performing a union via the previously defined Weighted Union Find implementation.

```java
public static double[] getThreshold(int N, int SAMPLES) {
    Random rand = new Random();
    double size = N*N;
    double[] thresholds = new double[SAMPLES];

    for (int i = 0; i < SAMPLES; i++) {
        Percolation perc = new Percolation(N);

        while (true) {
            int row = rand.nextInt(N) + 1;
            int col = rand.nextInt(N) + 1;

            perc.open(row, col);

            if (perc.percolates()) {
                thresholds[i] = perc.getOpenCount() / size;
                break;
            }
        }

    }

    return thresholds;
}
```

To create the simulation to estimate the percolation threshold, I used a method setup similar to the one used in the last two sections. To get the threshold of an N x N grid, using a number of samples, I created the getThreshold method. I create a new grid, then generate the row and column coordinates to open randomly. The check to openness is done in the .open method, so we don't worry about that here. We check if it has percolated, and if it does we add the probability (open sites / grid size) to the list of samples. After this, we break from the while and continue the for loop to fill the sample array.

```java
public static double[] run(int N, int SAMPLES) {
    double[] thresholds = PercEstimator.getThreshold(N, SAMPLES);
    double mean = Util.mean(thresholds);
    double stdDev = Util.stdDev(thresholds);

    String out = String.format("Grid Size: %d\nThreshold: %,.4f\nstd Dev: %,.4f\n",
                               N, mean, stdDev);
    System.out.println(out);

    return new double[] {mean, stdDev};
}
```

Using this is the run method, which gets these samples, and calculates the mean (no outlier removal) and standard deviation.

```java
Run | Debug
public static void main(String[] args) {
    int SIZE = 500;
    int SAMPLES = 500;
    int STEPS = 10;
    int START = 10;


    int arraySize = (SIZE - START) / STEPS;
    Integer[] sizes = new Integer[arraySize];
    for (int i = 0; i < arraySize; i++) {
        sizes[i] = START + i * STEPS;
    }

    Double[] thresholds = new Double[sizes.length];
    Double[] stdDevs = new Double[sizes.length];
    String[][] out = new String[sizes.length][3];

    for (int i = 0; i < sizes.length; i++) {
        double[] result = run(sizes[i], SAMPLES);
        thresholds[i] = result[0];
        stdDevs[i] = result[1];
    }

    for (int i = 0; i < sizes.length; i++) {
        out[i] = new String[]{String.format("%d", sizes[i]),
                              String.format("%,.5f", thresholds[i]),
                              String.format("%,.5f", stdDevs[i])};
    }
    Util.writeCSV(path:"src/graphs/perc/runs.csv", new String[]{"Sizes", "Thresholds", "Standard Dev"}, out);



    Plotter<Integer, Double> plt = new Plotter<>(path:"perc/thresholds.png",
                                                 x_label:"Grid Size",
                                                 y_label:"probability",
                                                 Plotter.Type.LINE,
                                                 title:"Percolation thresholds of varying grid sizes");

    plt.add(sizes, thresholds, label:"Threshold");
    //plt.add(sizes, stdDevs, "Standard Deviation");
    plt.plot();
}
```

With this, I can simply create a list of grid sizes to test, and define my sample size. A simple for loop and two arrays to hold the results is needed, and that would give me the basis for the testing.

I had decided to graph the results, as well as writing them to a CSV to extract into tabular format.

3.2. Results

Overall, the estimated threshold probability p* seems to stay at p = 0.59, fluctuating only slightly as the grid size increases.

As we can see from the below table of sizes, the size of p* fluctuates but stays at approximately 0.59 for relatively large grids. The standard deviation however, decreases as the size grows, meaning the precision of the percolation threshold estimation also increases.

Size (N)	Threshold (p)	Standard Dev
50	0.59165	0.0236
100	0.58983	0.01804
150	0.59319	0.01315
200	0.5935	0.00819
250	0.59146	0.00766
300	0.59478	0.00688
350	0.59421	0.00643
400	0.59271	0.00563
450	0.59344	0.00473
500	0.59279	0.00443
550	0.59277	0.00551
600	0.59184	0.00452
650	0.59291	0.00433
700	0.59352	0.00401
750	0.5925	0.00388
800	0.59334	0.00295
900	0.59303	0.00371
950	0.59331	0.0026

A number of other readings at steps of 10 in grid size, up to a size of 990 were taken. This is a slice of that dataset intended for easy viewing.

Within the entirety of the runs performed in finding percolation threshold (p), $0.58800 \leq p \leq 0.59638$. Also $p > 0.58881 \ \forall \ N > 130$.

This leads me to believe that the percolation threshold, p, for large enough grids lies in the area of $0.588 \leq p < 0.600$. From what I've gathered from research, this is to be expected [1].

References

1. [1] M. E. Newman and R. M. Ziff, “Efficient Monte Carlo algorithm and high-precision results for percolation,” *Physical Review Letters*, vol. 85, no. 19, pp. 4104–4107, 2000. doi:10.1103/physrevlett.85.4104