Programming Assignment 01

1DV516

Name: Tadj Cazaubon
Student: tc222gf
Email: tc222gf@gmail.lnu.se

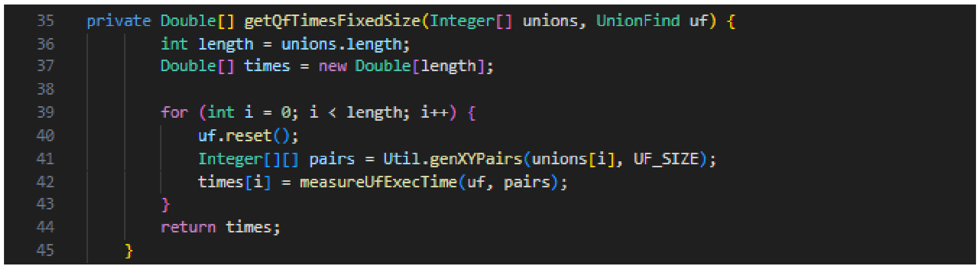
Problem 4

For this problem, I implemented the Quick Find, and (height-based) Weighted Union Find, with no path compression. To determine the growth of each, I split them into two groups, one for the data presented in the lecture slides, then one for larger arrays (100,000 and 1,000,000 elements respectively).

Firstly, I created the Timeit module, which acts similarly to python's timeit. It can dynamically take in a method from an Object with any number of arguments, and time its execution in nanoseconds, microseconds, etc.

Then the Plotter, which calls the Python script to plot the graph of the runs. At first, I dynamically generated the number of unions to perform, but I thought it'd be better for testing to just have a fixed, relatively small range for testing.

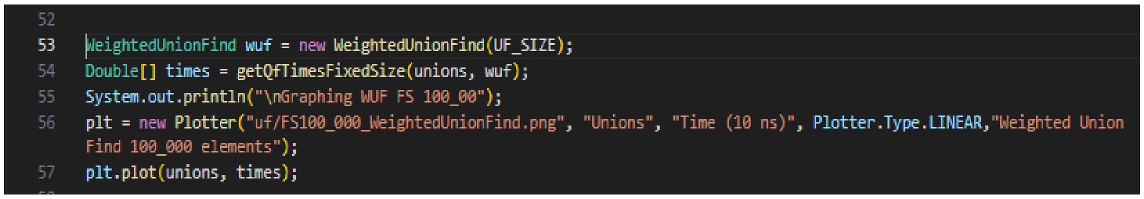
The way in which it works is that for each union N, a random N-long array of X, Y pairs to be unioned is generated. The average time over a pre-determined number of samples is taken and saved to the time array.



```java
private Double[] getQfTimesFixedSize(Integer[] unions, UnionFind uf) {
        int length = unions.length;
        Double[] times = new Double[length];

        for (int i = 0; i < length; i++) {
            uf.reset();
            Integer[][] pairs = Util.genXYPairs(unions[i], UF_SIZE);
            times[i] = measureUfExecTime(uf, pairs);
        }
        return times;
    }
```

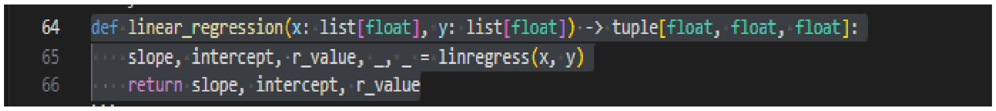
When the execution time is recorded, outliers are also removed in sampleMean() via the IQR to remove random jitter from big processor state changes. These were prevalent whenever I used my computer for something else while testing at first. This is over 100 samples per X reading.

Once the times are recorded for each number of unions, the times are sent to the Plotter. I took inspiration from Matplotlib, where I have a Plotter object which can change state and be reused. The path, x and y labels, titles, and expected graph type can be specified on instantiation.



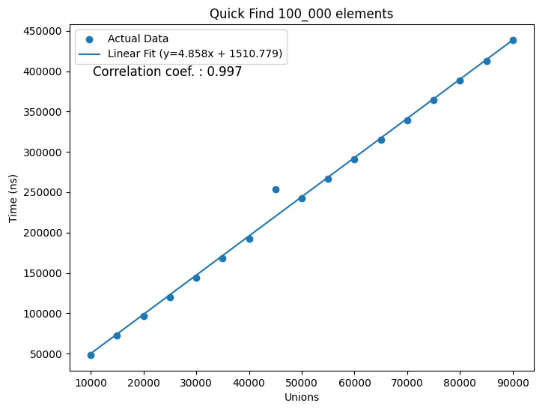
```java

WeightedUnionFind wuf = new WeightedUnionFind(UF_SIZE);
Double[] times = getQfTimesFixedSize(unions, wuf);
System.out.println("\nGraphing WUF FS 100_00");
plt = new Plotter("uf/FS100_000_WeightedUnionFind.png", "Unions", "Time (10 ns)", Plotter.Type.LINEAR,"Weighted Union Find 100_000 elements");
plt.plot(unions, times);
```

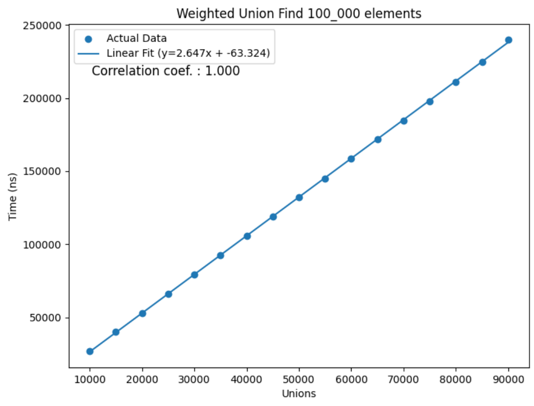
Once created, you can call plot (), passing in two lists, the corresponding X and Y values. The does not only plot the graphs, but also performs linear regression to find the line-of-best-fit on the data provided, as well as the linear correlation coefficient to determine how closely it fits the calculated line.



```python
def linear_regression(x: list[float], y: list[float]) -> tuple[float, float, float]:
    slope, intercept, r_value, _, _ = linregress(x, y)
    return slope, intercept, r_value
```

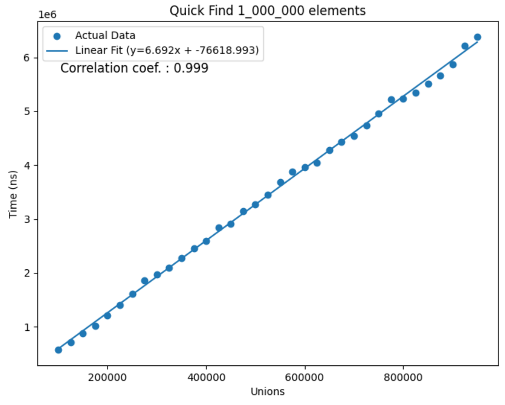
Union Find Results:

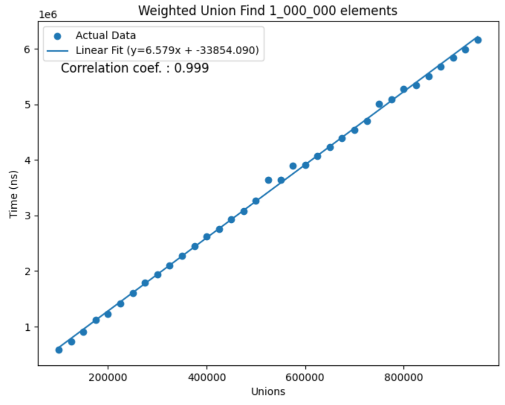




Here we can see that for up to a tree of 95000 elements, the Quick Find is a much faster growing function, with a greater slope, about 1.8 times that of the Weighted Union Find. The correlation coefficient for these is also very high for the subset of points chosen. Both are therefore linear functions, but the Weighted Union is growing much slower. This matches my expectations closely with what I've seen from the lectures.

The larger arrays produce a strange but somewhat understandable result. At first glance, the graphs seem almost identical, but a look at the slope reveals that the Weighted Union Find still retains a lower slope, though somewhat hard to see. The equations for each set of readings differ mostly by the y-intercept and have near perfect linear correlation.

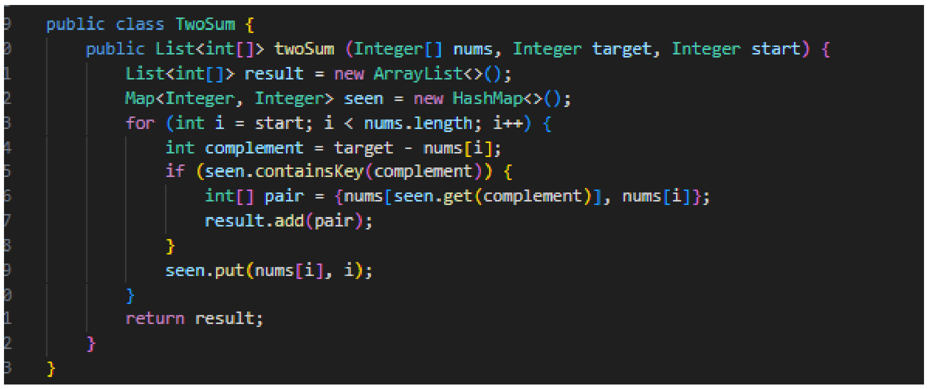




While I expected the Weighted find to continue to dominate speed-wise, I suppose this makes sense that with enough data, their positions should equalize somewhat, especially as the trees are not being flattened since I opted to not use path compression.

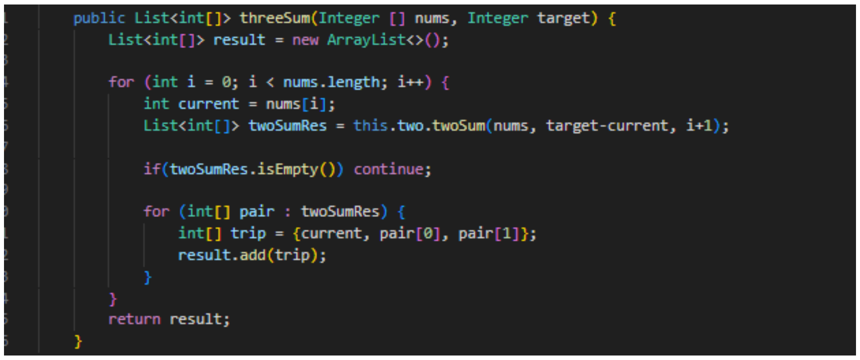
Problem 7

In implementing the cached 3-sum, I first implemented a cached 2-sum:



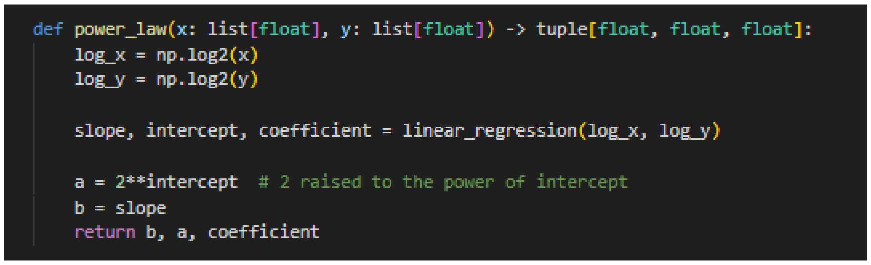
```
public class TwoSum {
    public List<int[]> twoSum (Integer[] nums, Integer target, Integer start) {
        List<int[]> result = new ArrayList<>();
        Map<Integer, Integer> seen = new HashMap<>();
        for (int i = start; i < nums.length; i++) {
            int complement = target - nums[i];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

With this, my search for any two numbers will no longer be quadratic, but linear. Now, I implement this within the 3-sum, which will take 3-sum searching from a cubic task, to a quadratic one theoretically.



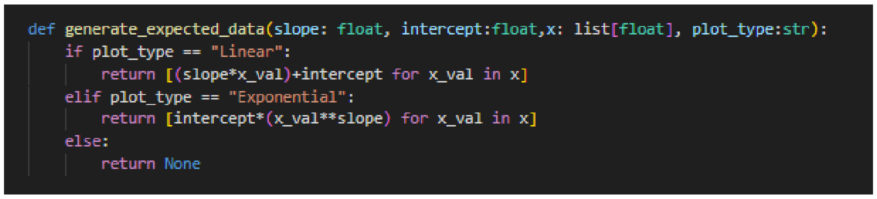
```
    public List<int[]> threeSum(Integer [] nums, Integer target) {
        List<int[]> result = new ArrayList<>();

        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;
    }
```

Graphing these two out is largely the same as with the Union Find, though now I specify that my data is theoretically "Exponential" in the Plotter. When the data is ingested by the script, it will now transform the data into the Log2 equivalent and perform linear regression on the log-log data.



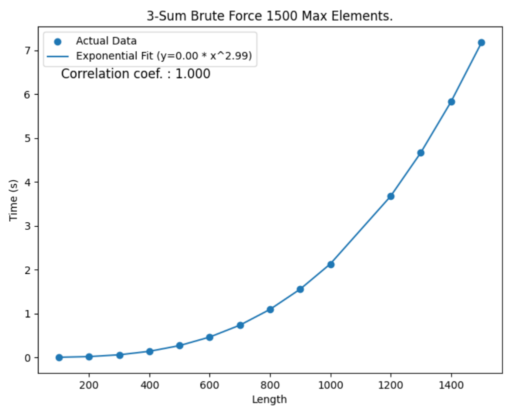
```
def power_law(x: list[float], y: list[float]) -> tuple[float, float, float]:
    log_x = np.log2(x)
    log_y = np.log2(y)

    slope, intercept, coefficient = linear_regression(log_x, log_y)

    a = 2**intercept  # 2 raised to the power of intercept
    b = slope
    return b, a, coefficient
```

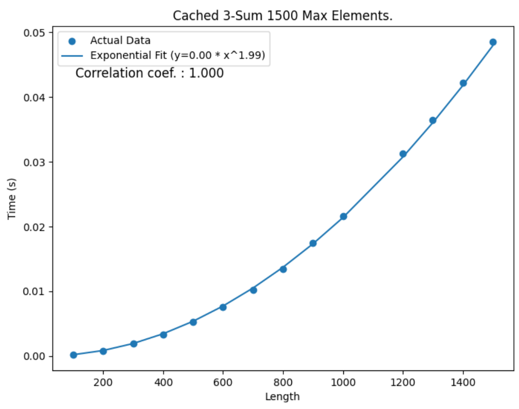
From this, we get the same data as the linear set above, but we can use this to create a power law and generate the expected data to compare to the real data. This function is crude but effective for small data ranges:



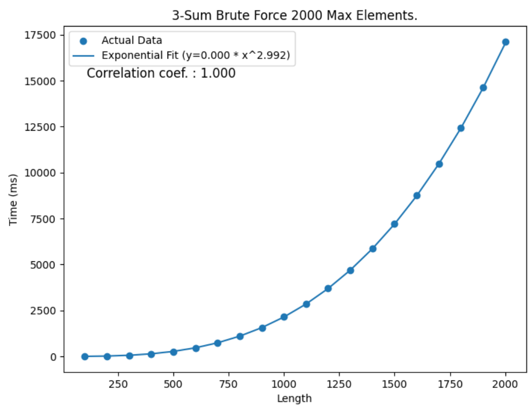
```python
def generate_expected_data(slope: float, intercept:float,x: list[float], plot_type:str):
    if plot_type == "Linear":
        return [(slope*x_val)+intercept for x_val in x]
    elif plot_type == "Exponential":
        return [intercept*(x_val**slope) for x_val in x]
    else:
        return None
```

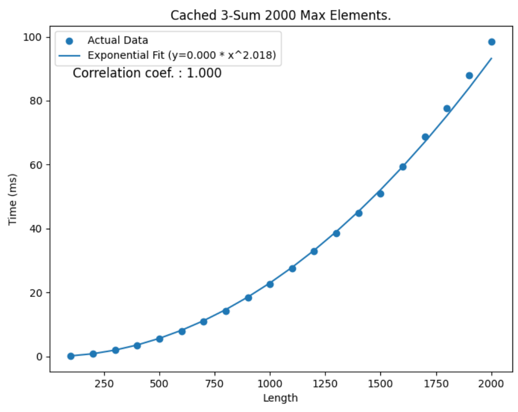
3-Sum Results:





A few things are immediately clear. The cached 3-sum variant takes much less time. Looking at the power law equations, we can see the slope of the brute-force variant is practically 3, and the cached variant has a slope of practically 2, meaning they are cubic and quadratic respectively. This is almost exactly what I would expect as an outcome, as the brute-force approach to 3-sum is cubic, whilst the cached implementation is expected to be quadratic. I also managed to get results for both implementations up to a maximum of 2000 elements:





Once again, the difference in the times is immense. The cached implementation once again has a slope of approximately 2, and the brute-force implementation, approximately 3. These functions are without any doubt cubic and quadratic respectively. I wanted to test for larger values to compare but couldn't stand waiting for the brute-force variant.