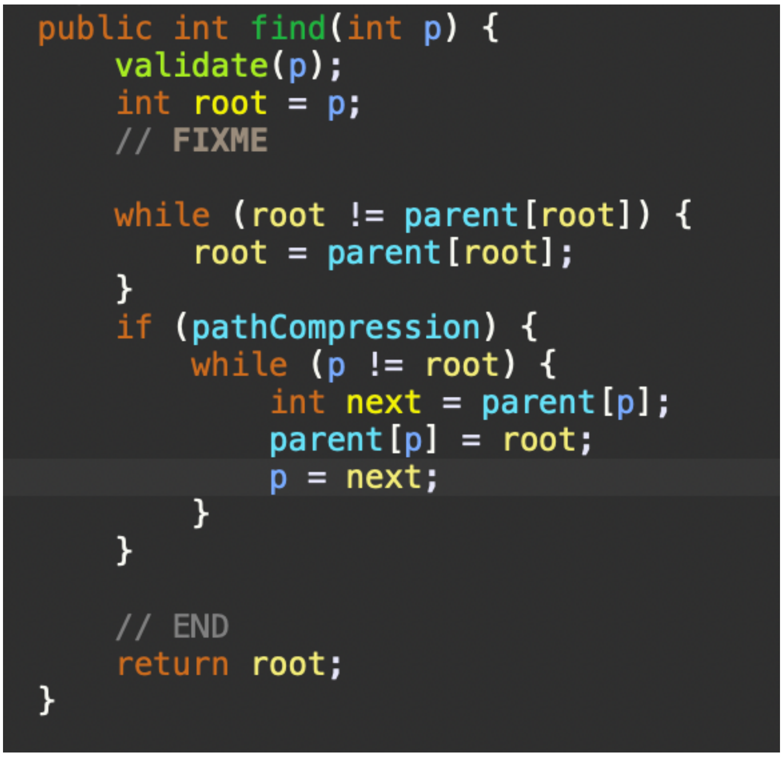
Assignment 4 - Program Structures and Algorithms Spring 2023(SEC - 1)

NAME: Paurush Batish **NUID: 002755631**

Your task is

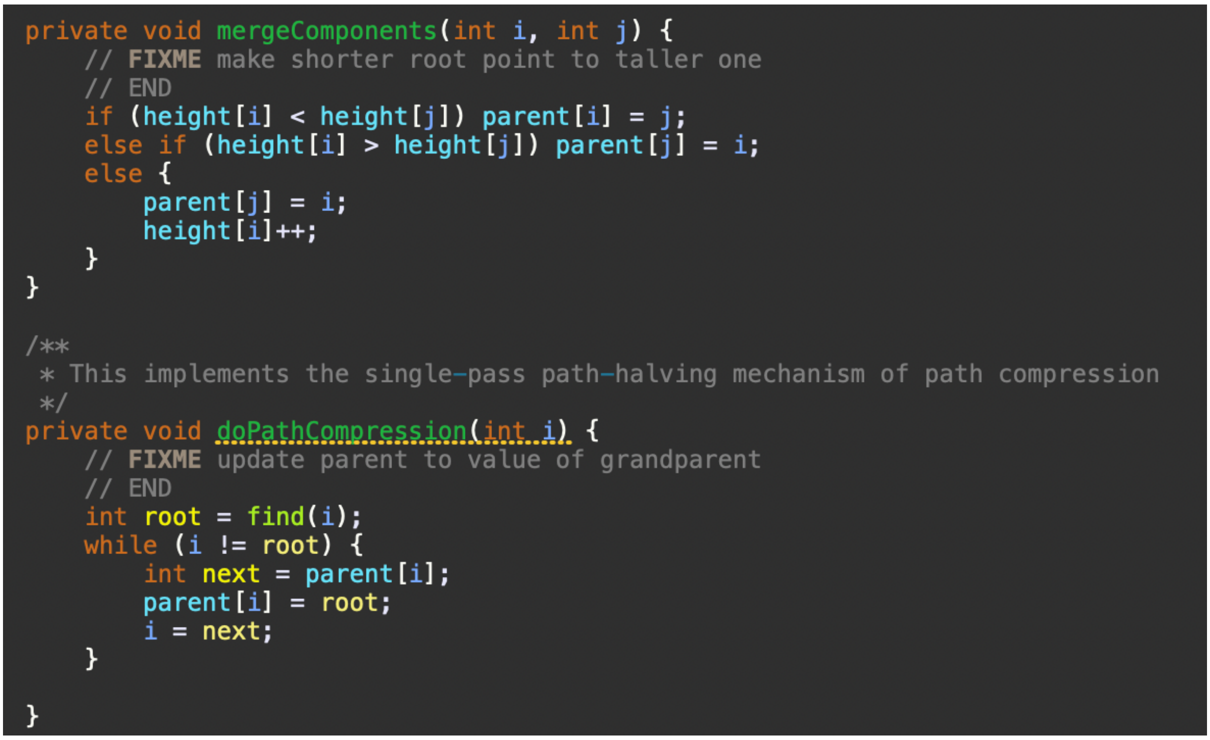
Step 1:
(a) Implement height-weighted Quick Union with Path Compression. For this, you will flesh out the class UF_HWQUPC. All you have to do is to fill in the sections marked with // TO BE IMPLEMENTED ... // ...END IMPLEMENTATION.



```
public int find(int p) {
    validate(p);
    int root = p;
    // FIXME

    while (root != parent[root]) {
        root = parent[root];
    }
    if (pathCompression) {
        while (p != root) {
            int next = parent[p];
            parent[p] = root;
            p = next;
        }
    }

    // END
    return root;
}
```

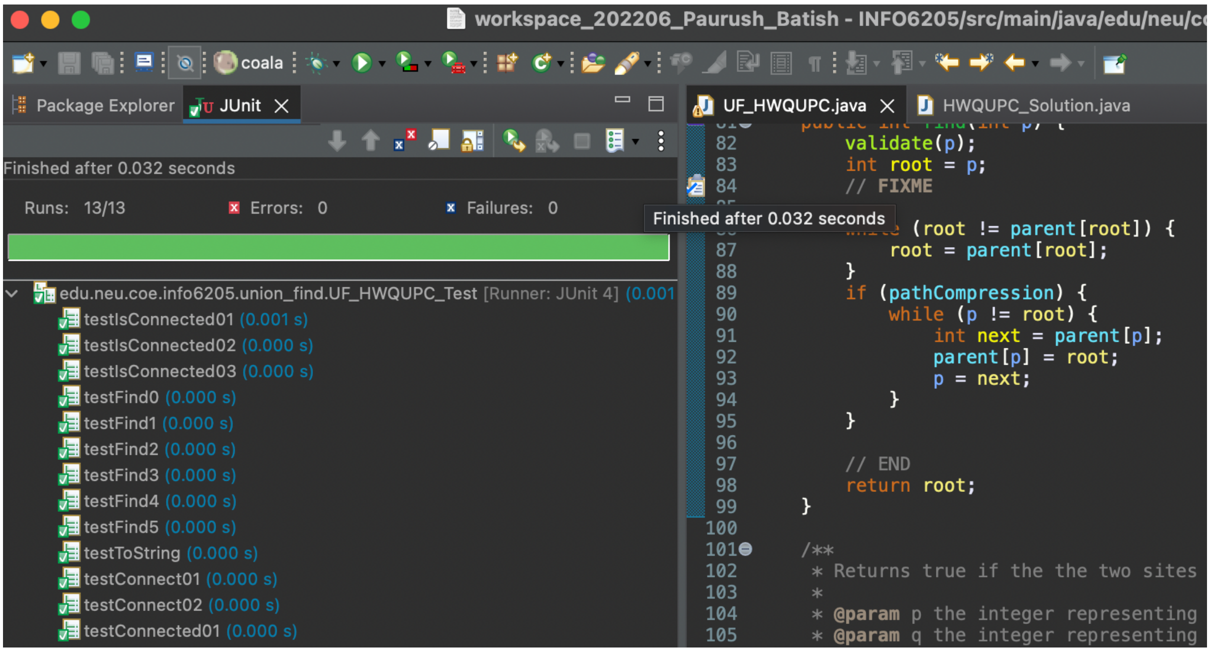


```
private void mergeComponents(int i, int j) {
    // FIXME make shorter root point to taller one
    // END
    if (height[i] < height[j]) parent[i] = j;
    else if (height[i] > height[j]) parent[j] = i;
    else {
        parent[j] = i;
        height[i]++;
    }
}

/**
 * This implements the single-pass path-halving mechanism of path compression
 */
private void doPathCompression(int i) {
    // FIXME update parent to value of grandparent
    // END
    int root = find(i);
    while (i != root) {
        int next = parent[i];
        parent[i] = root;
        i = next;
    }

}
```

(b) Check that the unit tests for this class all work. You must show "green" test results in your submission (screenshot is OK).


workspace_202206_Paurush_Batish - INFO6205/src/main/java/edu/neu/c
coala
Package Explorer
JUnit
UF_HWQUPC.java
HWQUPC_Solution.java

Finished after 0.032 seconds

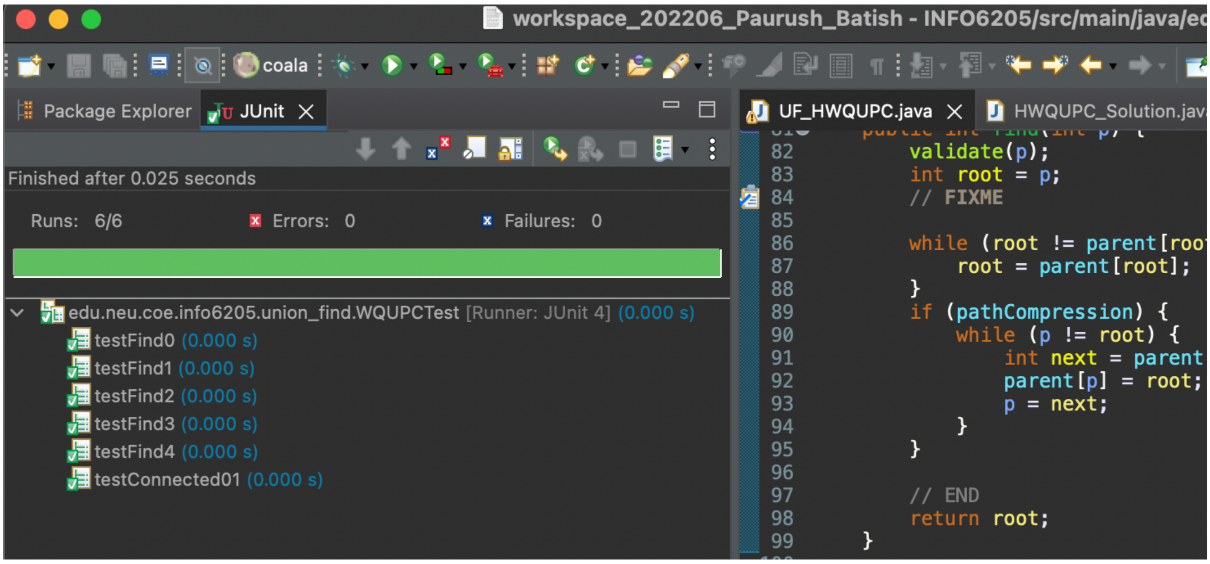
Runs: 13/13   Errors: 0   Failures: 0

```
edu.neu.coe.info6205.union_find.UF_HWQUPC_Test [Runner: JUnit 4] (0.001
    testIsConnected01 (0.001 s)
    testIsConnected02 (0.000 s)
    testIsConnected03 (0.000 s)
    testFind0 (0.000 s)
    testFind1 (0.000 s)
    testFind2 (0.000 s)
    testFind3 (0.000 s)
    testFind4 (0.000 s)
    testFind5 (0.000 s)
    testToString (0.000 s)
    testConnect01 (0.000 s)
    testConnect02 (0.000 s)
    testConnected01 (0.000 s)
```

```
        validate(p);
        int root = p;
        // FIXME
Finished after 0.032 seconds
              (root != parent[root]) {
            root = parent[root];
        }
        if (pathCompression) {
            while (p != root) {
                int next = parent[p];
                parent[p] = root;
                p = next;
            }
        }

        // END
        return root;
    }

    /**
     * Returns true if the the two sites
     *
     * @param p the integer representing
     * @param q the integer representing
```



workspace_202206_Paurush_Batish - INFO6205/src/main/java/ed

Finished after 0.025 seconds

Runs: 6/6   Errors: 0   Failures: 0

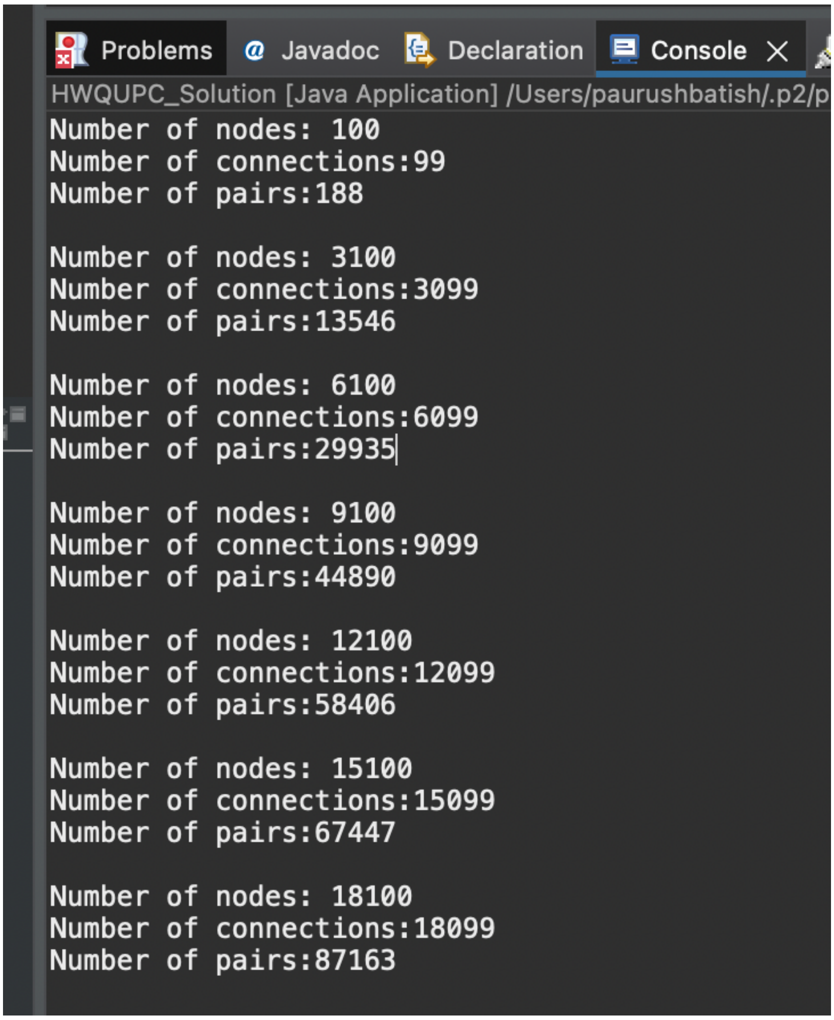
```
edu.neu.coe.info6205.union_find.WQUPCTest [Runner: JUnit 4] (0.000 s)
    testFind0 (0.000 s)
    testFind1 (0.000 s)
    testFind2 (0.000 s)
    testFind3 (0.000 s)
    testFind4 (0.000 s)
    testConnected01 (0.000 s)
```

```java
    public int find(int p) {
        validate(p);
        int root = p;
        // FIXME

        while (root != parent[roo
            root = parent[root];
        }
        if (pathCompression) {
            while (p != root) {
                int next = parent
                parent[p] = root;
                p = next;
            }
        }

        // END
        return root;
    }
```

Step 2:
Using your implementation of UF_HWQUPC, develop a UF ("union-find") client that takes an integer value n from the command line to determine the number of "sites." Then generates random pairs of integers between 0 and n-1, calling connected() to determine if they are connected and union() if not. Loop until all sites are connected then print the number of connections generated. Package your program as a static method count() that takes n as the argument and returns the number of connections; and a main() that takes n from the command line, calls count() and prints the returned value. If you prefer, you can create a main program that doesn't require any input and runs the experiment for a fixed set of n values. Show evidence of your run(s).



```
HWQUPC_Solution [Java Application] /Users/paurushbatish/.p2/p
Number of nodes: 100
Number of connections:99
Number of pairs:188

Number of nodes: 3100
Number of connections:3099
Number of pairs:13546

Number of nodes: 6100
Number of connections:6099
Number of pairs:29935

Number of nodes: 9100
Number of connections:9099
Number of pairs:44890

Number of nodes: 12100
Number of connections:12099
Number of pairs:58406

Number of nodes: 15100
Number of connections:15099
Number of pairs:67447

Number of nodes: 18100
Number of connections:18099
Number of pairs:87163
```

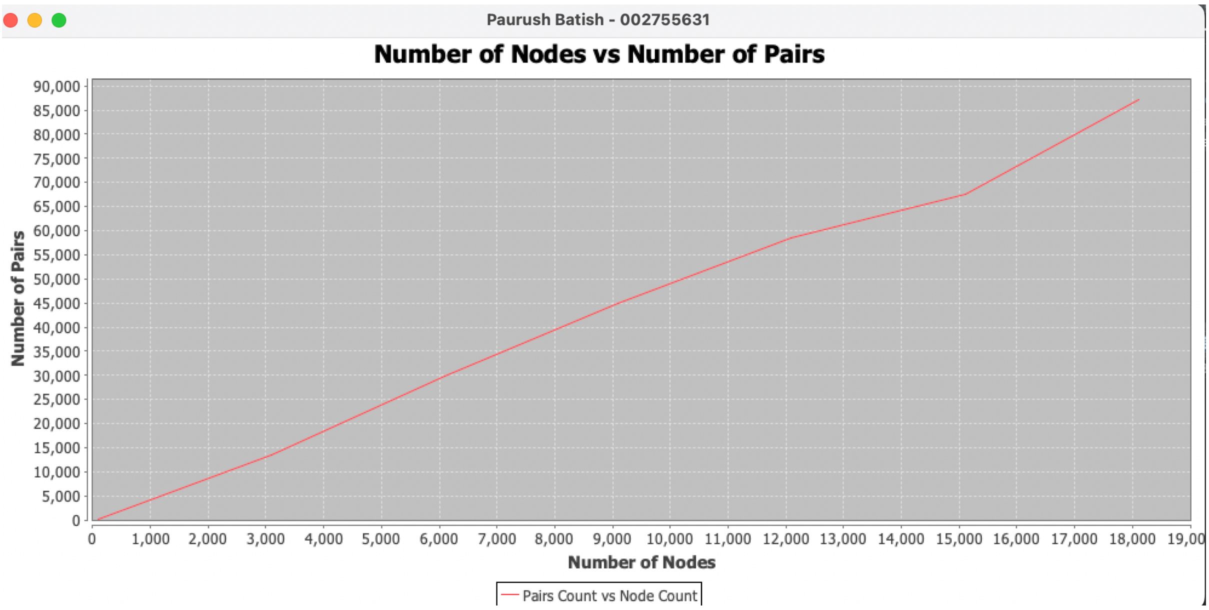
Step 3:
Determine the relationship between the number of objects (*n*) and the number of pairs (*m*) generated to accomplish this (i.e. to reduce the number of components from *n* to 1). Justify your conclusion in terms of your observations and what you think might be going on.

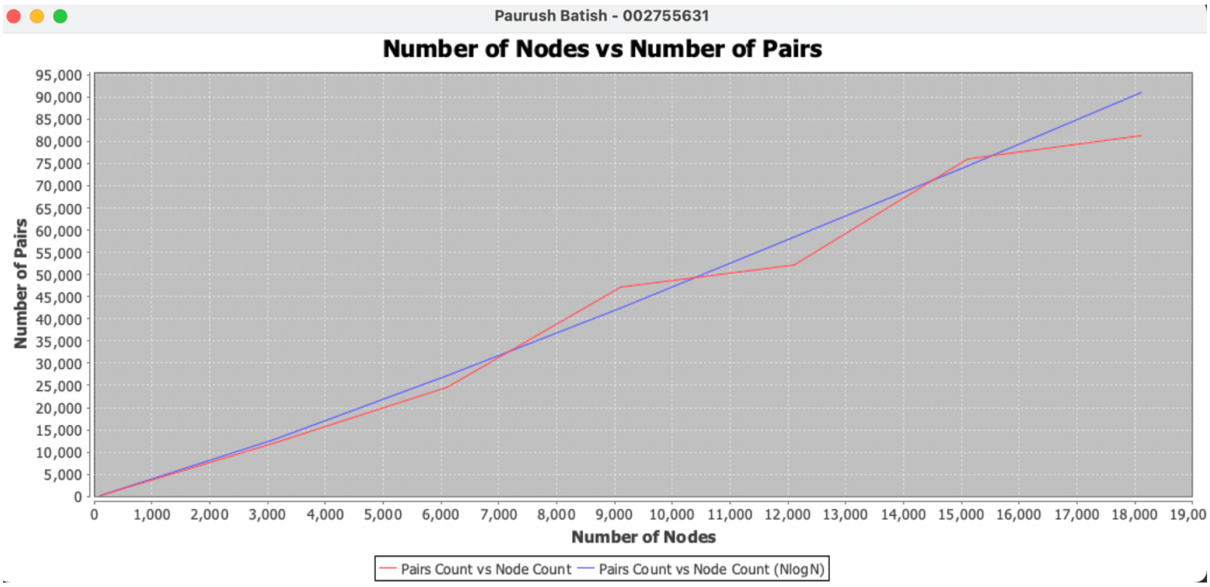
NOTE: although I'm not going to tell you in advance what the relationship is, I can assure you that it is a *simple* relationship.

Don't forget to follow the submission guidelines. And to use sufficient (and sufficiently large) different values of n.

Conclusion –

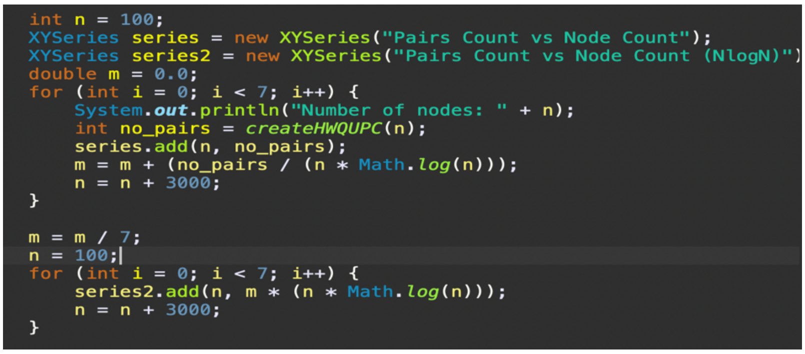
Based on the mathematical relations and the graph values the relationship is appearing to be having a complexity of $n\log(\mathrm{n})$. Let's see the proof of that.





The blue line represents the relation between node count and pair count if we choose the relation as y = m * n*log(n)
Where m is taken as the ratio.

How we calculated it →



```
int n = 100;
XYSeries series = new XYSeries("Pairs Count vs Node Count");
XYSeries series2 = new XYSeries("Pairs Count vs Node Count (NlogN)")
double m = 0.0;
for (int i = 0; i < 7; i++) {
    System.out.println("Number of nodes: " + n);
    int no_pairs = createHWQUPC(n);
    series.add(n, no_pairs);
    m = m + (no_pairs / (n * Math.log(n)));
    n = n + 3000;
}

m = m / 7;
n = 100;
for (int i = 0; i < 7; i++) {
    series2.add(n, m * (n * Math.log(n)));
    n = n + 3000;
}
```