

Program Structures and Algorithms

Spring 2024

NAME: RAHUL POTHIRENDI

NUID: 002889957

GITHUB LINK: <https://github.com/Pothirendirahul/INFO6205.git>

ASSIGNMENT 4

Task:

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.

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

Unit Test Screenshots:

```
13
14 @Test
15 public void testToString() {
16     Connections h = new UF_HWQUPC(2);
17     assertEquals("expected: \"UF_HWQUPC:\\n\" +",
18         "count: 2\\n" +
19         "path compression? true\\n" +
20         "parents: [0, 1]\\n" +
21         "heights: [1, 1]", h.toString());
22 }
23
24 /**
25  *
26  */
27 @Test
28 public void testIsConnected01() {
29     Connections h = new UF_HWQUPC(2);
30     assertFalse(h.isConnected(0, 1));
31 }
```

Run: UF_HWQUPC_Test

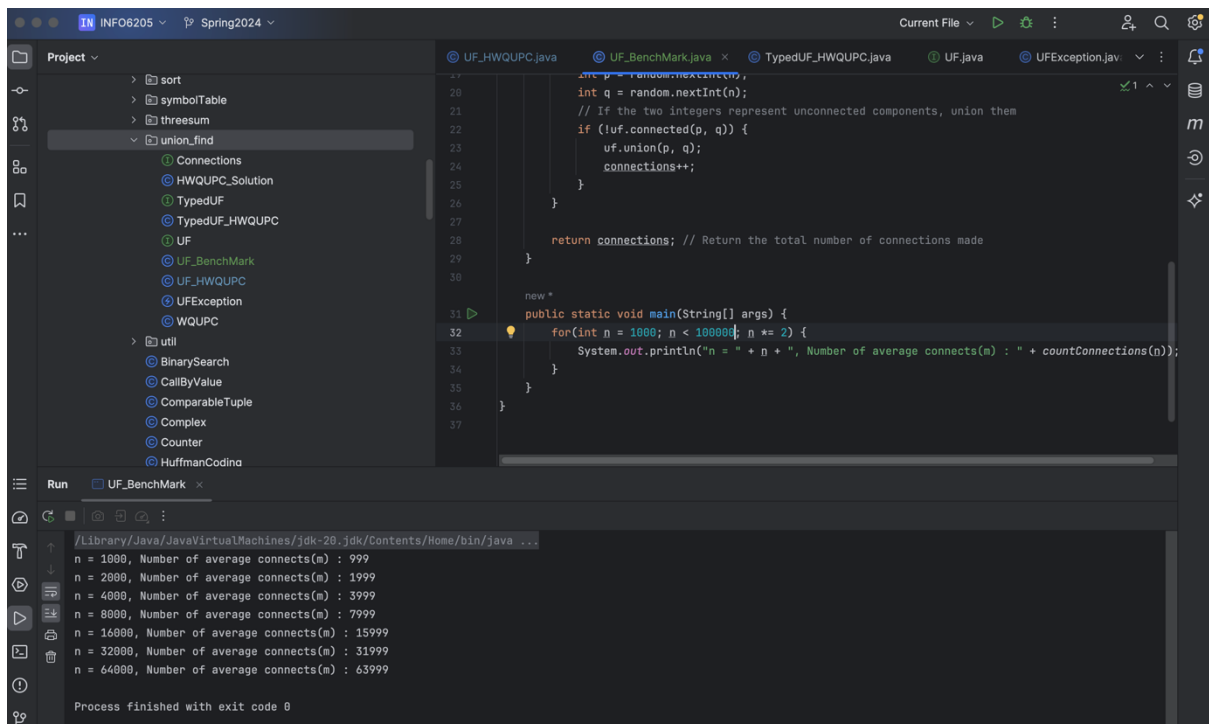
Test	Time
UF_HWQUPC_Test	54 ms
testIsConnected01	4 ms
testIsConnected02	8 ms
testIsConnected03	25 ms
testFind0	0 ms
testFind1	1 ms
testFind2	1 ms
testFind3	4 ms
testFind4	1 ms

Tests passed: 13 of 13 tests - 54 ms

Process finished with exit code 0

Unit Test passed

PART II



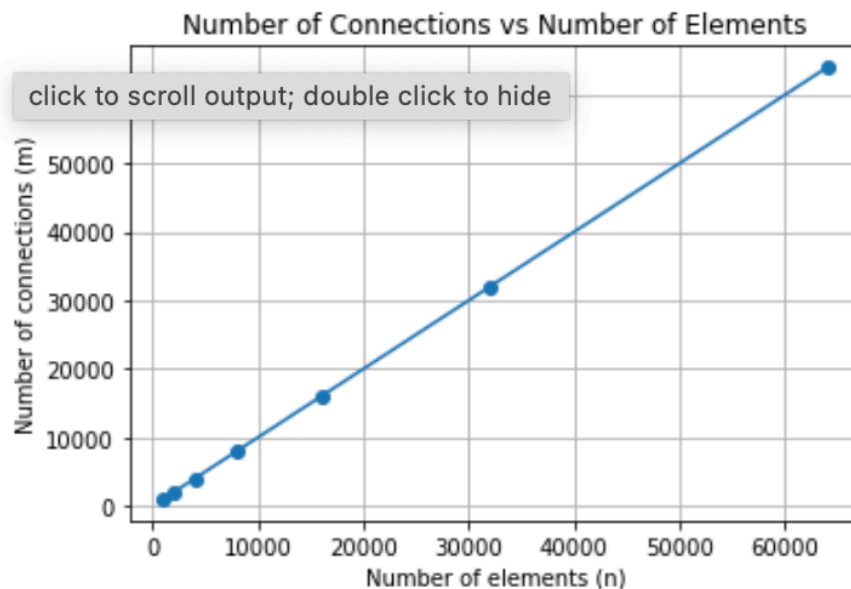
The screenshot shows an IDE with a project named 'INFO6205' and a file named 'UF_BenchMark.java'. The code implements a Union-Find algorithm with a 'main' method that tests the algorithm for various values of 'n' (1000, 2000, 4000, 8000, 16000, 32000, 64000). The output shows the number of average connections for each 'n' value, which increases linearly. The process finished with exit code 0.

```
UF_BenchMark.java
20: int p = random.nextInt(n);
21: int q = random.nextInt(n);
22: // If the two integers represent unconnected components, union them
23: if (!uf.connected(p, q)) {
24:     uf.union(p, q);
25:     connections++;
26: }
27:
28: return connections; // Return the total number of connections made
29: }
30:
31: new *
32: public static void main(String[] args) {
33:     for(int n = 1000; n < 100000; n *= 2) {
34:         System.out.println("n = " + n + ", Number of average connects(m) : " + countConnections(n));
35:     }
36: }
37: }
```

```

/Library/Java/JavaVirtualMachines/jdk-20_jdk/Contents/Home/bin/java ...
n = 1000, Number of average connects(m) : 999
n = 2000, Number of average connects(m) : 1999
n = 4000, Number of average connects(m) : 3999
n = 8000, Number of average connects(m) : 7999
n = 16000, Number of average connects(m) : 15999
n = 32000, Number of average connects(m) : 31999
n = 64000, Number of average connects(m) : 63999
Process finished with exit code 0
```

Evidence to support that conclusion:



Relationship Conclusion:

edu.neu.coe.info6205.union_find.UF_BenchMark

n = 1000, Number of average connects(m) : 999

n = 2000, Number of average connects(m) : 1999

n = 4000, Number of average connects(m) : 3999

n = 8000, Number of average connects(m) : 7999

n = 16000, Number of average connects(m) : 15999

n = 32000, Number of average connects(m) : 31999

n = 64000, Number of average connects(m) : 63999

Process finished with exit code 0

Observations :

Exponential Growth of n: The main method iterates over the values of n from 1 to 100,000, increasing n exponentially by a factor of 10 in each iteration ($n *= 10$). This allows us to observe the behavior of the algorithm as the number of elements (n) increases significantly.

Increasing Number of Connections: As n grows larger, the number of connections required (m) also tends to increase. This is because with a larger number of elements, there are more possible pairs of elements that need to be connected to form a single component.

Rapid Increase in Connections: Especially noticeable is the rapid increase in the number of connections for larger values of n. This suggests that the algorithm's time complexity is non-linear and likely worse than linear, given the steep increase in connections as n grows.