

**Name:** Mani Charan Reddy Loka

**NUID:** 002727403

**Task:**

Step 1) Implement height-weighted Quick Union with Path Compression

**find method:**

```
81 public int find(int p) {
82     validate(p);
83     int root = p;
84     // FIXME
85     while(root!=parent[root]) {
86         root=parent[root];
87     }
88     if(pathCompression) {
89         while(p!=root) {
90             //int newp = parent[p];
91             doPathCompression(p);
92             p=parent[p];
93         }
94     }
95     // END |
96     return root;
97 }
```

**mergeComponents method:**

```
180
181 private void mergeComponents(int i, int j) {
182     int rootI=find(i);
183     int rootJ=find(j);
184     if(rootI==rootJ) return;
185     if(height[rootI]<height[rootJ]) {
186         parent[rootI]=rootJ;
187         height[rootJ]+=height[rootI];
188     }
189     else {
190         parent[rootJ]=rootI;
191         height[rootI]+=height[rootJ];
192     }
193     }
194     //count--;
195     // FIXME make shorter root point to taller one
196     // END
197 }
```

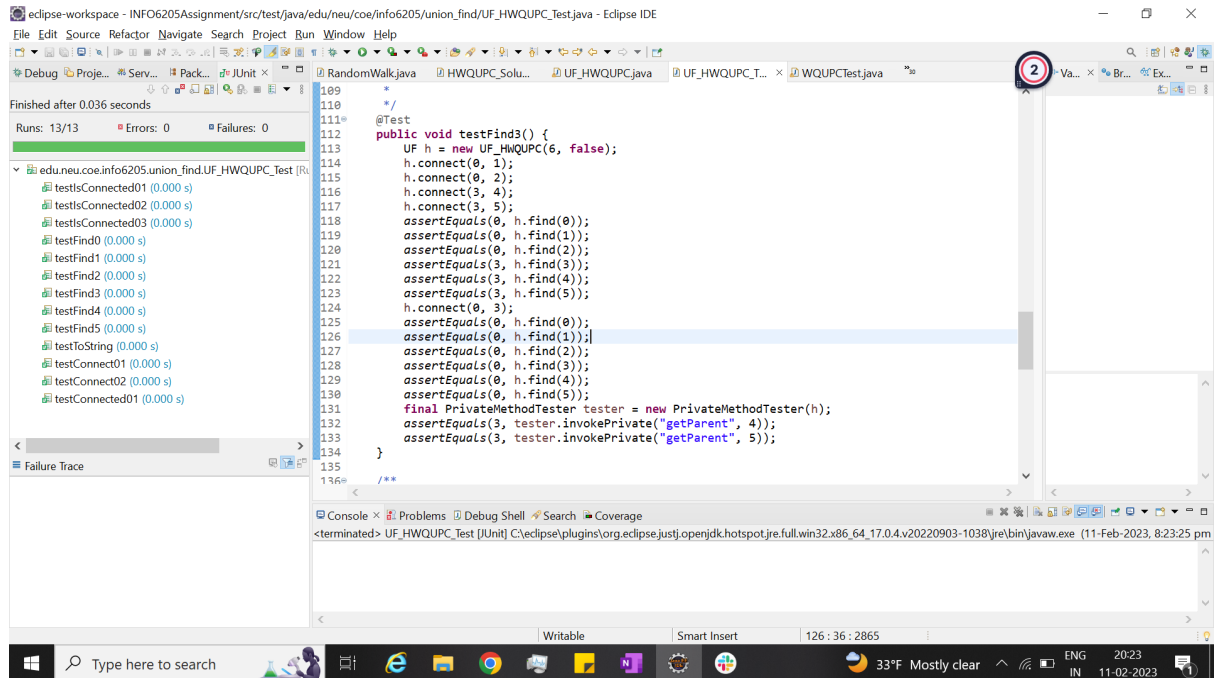
**doPathCompression method:**

```
private void doPathCompression(int i) {
    // FIXME update parent to value of grandparent
    //parent[i]=parent[parent[i]];
    parent[i]=getParent(getParent(i));
    // END
}
```

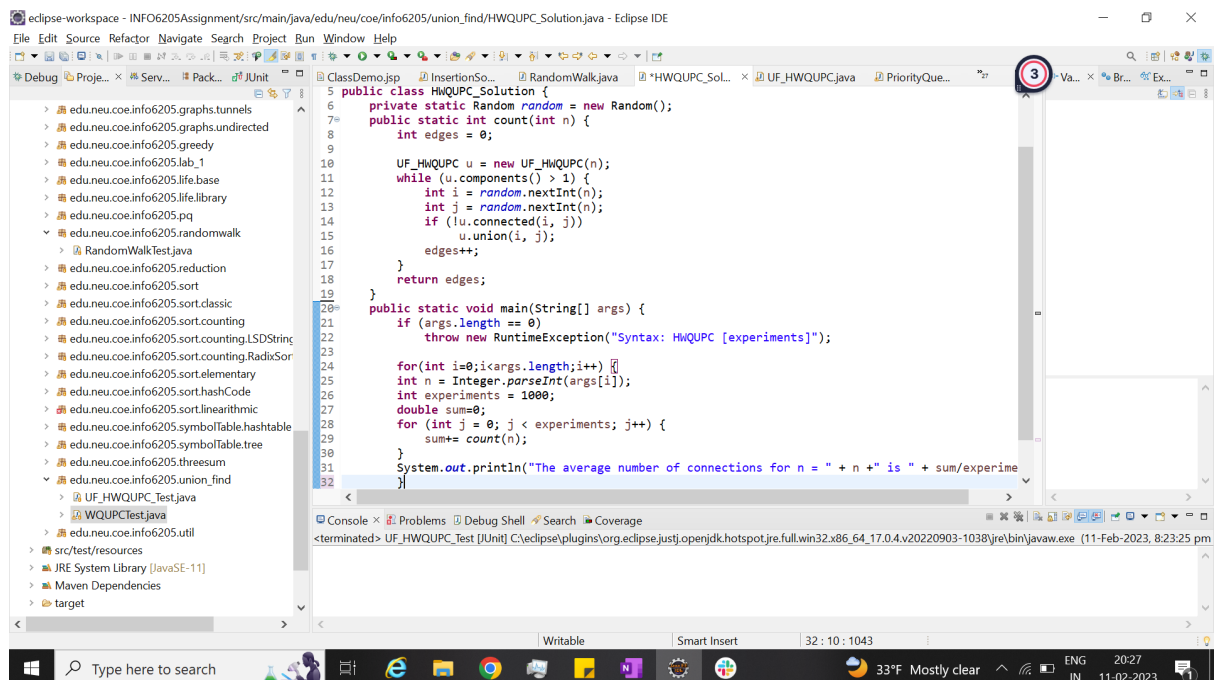
# Program Structures and Algorithms Spring 2023(Section - 1)

## Unit Test Screenshots:

### 1)UF\_HWQUPC\_Test.java

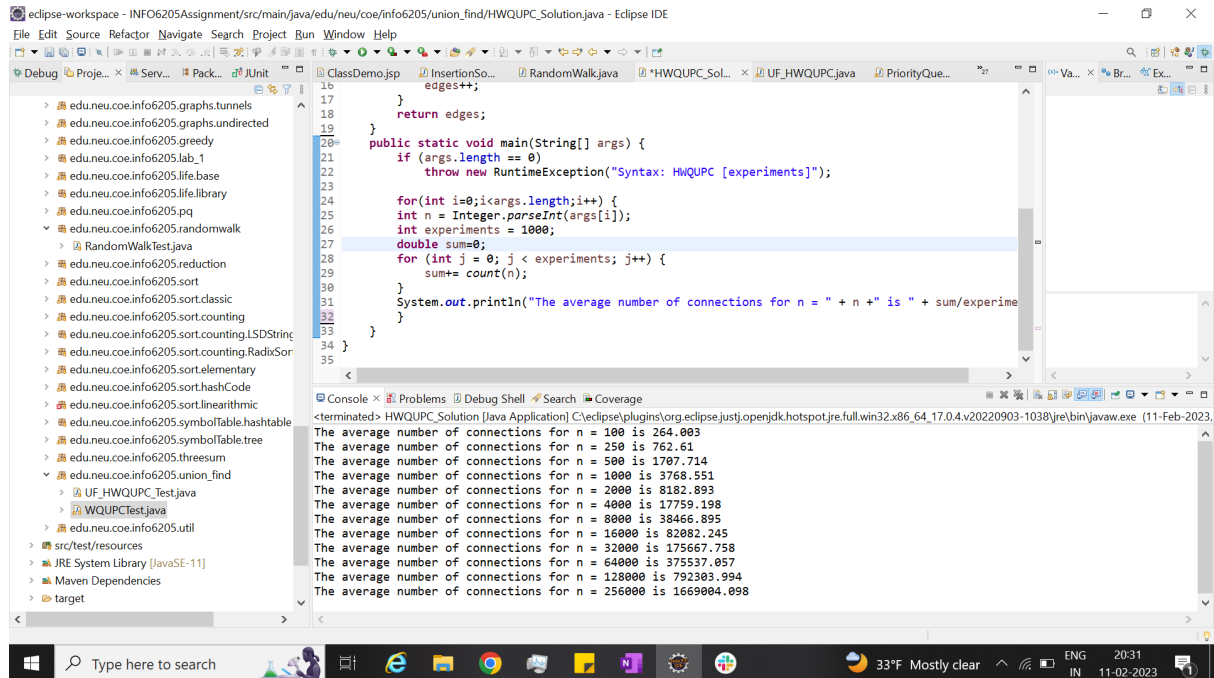


### Step 2)Using your implementation of UF\_HWQUPC, develop a UF ("union-find") client



## Program Structures and Algorithms Spring 2023(Section - 1)

### Output:



The screenshot shows the Eclipse IDE with the file `HWQUPC_Solution.java` open. The code defines a `main` method that takes an array of arguments. It checks if the number of arguments is zero, and if so, it throws a `RuntimeException` with the message "Syntax: HWQUPC [experiments]". Otherwise, it iterates over the arguments, parsing each as an integer `n` and performing a series of operations to calculate the average number of connections for that `n`. The console output shows the results for `n` values from 100 to 256000, with the average number of connections increasing as `n` increases.

```
public static void main(String[] args) {  
    if (args.length == 0) {  
        throw new RuntimeException("Syntax: HWQUPC [experiments]");  
    }  
    for (int i = 0; i < args.length; i++) {  
        int n = Integer.parseInt(args[i]);  
        int experiments = 1000;  
        double sum = 0;  
        for (int j = 0; j < experiments; j++) {  
            sum += count(n);  
        }  
        System.out.println("The average number of connections for n = " + n + " is " + sum/experiments);  
    }  
}
```

Console Output:

```
<terminated> HWQUPC_Solution [Java Application] C:\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe (11-Feb-2023).  
The average number of connections for n = 100 is 264.003  
The average number of connections for n = 250 is 762.61  
The average number of connections for n = 500 is 1707.714  
The average number of connections for n = 1000 is 3768.551  
The average number of connections for n = 2000 is 8182.893  
The average number of connections for n = 4000 is 17759.198  
The average number of connections for n = 8000 is 38466.895  
The average number of connections for n = 16000 is 82082.245  
The average number of connections for n = 32000 is 175667.758  
The average number of connections for n = 64000 is 375537.057  
The average number of connections for n = 128000 is 792303.994  
The average number of connections for n = 256000 is 1659004.098
```

Table showing no. of times connected method being called as against the input n

Objects(n)	Pairs(m)
100	263.03
250	760.7
500	1701.71
1000	3728.77
2000	8230.46
4000	17694.12
8000	38430.47
16000	82193.73
32000	175634.79
64000	374728.17
128000	790186.53
256000	1659293.67

Program Structures and Algorithms  
Spring 2023(Section - 1)

**Relationship Conclusion:**

The number of pairs generated to connect all the sites is linearly proportional to the log scale of no. of sites. I.e.,  $m$  is proportional to  $n \log n$ .

By averaging out the constant factor, it is found that the average constant factor is 0.37

Objects(n)	Pairs(m)	$n \cdot \log(n, 2)$	Average		$0.37 \cdot n \cdot \log n$
100	263.03	664.385619	0.3958995988		245.822679
250	760.7	1991.446071	0.3819837308		736.8350463
500	1701.71	4482.892142	0.3796009241		1658.670093
1000	3728.77	9965.784285	0.3741572056		3687.340185
2000	8230.46	21931.56857	0.3752791313		8114.680371
4000	17694.12	47863.13714	0.3696815766		17709.36074
8000	38430.47	103726.2743	0.370498895		38378.72148
16000	82193.73	223452.5486	0.3678352766		82677.44297
32000	175634.79	478905.0971	0.3667423693		177194.8859
64000	374728.17	1021810.194	0.3667297235		378069.7719
128000	790186.53	2171620.388	0.3638695484	Average factor	803499.5437
256000	1659293.67	4599240.777	0.3607755607	0.3727544617	1701719.087

Graph of  $0.37n \log n$  and  $m$ , as against  $n$ .

number of objects (n) v/s the number of pairs (m) and  $0.37 \cdot n \cdot \log n$

