Assignment 4

UF_HWQUPC code changes

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

    if(pathCompression){
        doPathCompression(p);
        root = parent[root];
    }
    else {
        while(root != parent[root]){
            root = parent[root];
        }
    }

    return root;
}
```

```java
private void mergeComponents(int i, int j) {
    // FIXME make shorter root point to taller one
    // END

    if (i==j) return;
    else if (height[i] == height[j]) {
        parent[j]=i;
        height[i]++;
    }
    else if(height[i] < height[j]) {
        parent[i] = j;
    }
    else {
        parent[j] = i;
    }

}
```

```java
private void doPathCompression(int i) {
    // FIXME update parent to value of grandparent
    // END

    while(i != parent[i]){
        parent[i] = parent[parent[i]];
        i = parent[i];
    }

}
```

```
/**
* Original code:
* Copyright © 2000–2017, Robert Sedgewick and Kevin Wayne.
* <p>
```

```
* Modifications:
* Copyright (c) 2017. Phasmid Software
*/
package edu.neu.coe.info6205.union_find;

import java.util.Arrays;

/**
* Height-weighted Quick Union with Path Compression
*/
public class UF_HWQUPC implements UF {
    /**
     * Ensure that site p is connected to site q,
     *
     * @param p the integer representing one site
     * @param q the integer representing the other site
     */
    public void connect(int p, int q) {
        if (!isConnected(p, q)) union(p, q);
    }

    /**
     * Initializes an empty union-find data structure with {@code n} sites
     * {@code 0} through {@code n-1}. Each site is initially in its own
     * component.
     *
     * @param n                  the number of sites
     * @param pathCompression whether to use path compression
     * @throws IllegalArgumentException if {@code n < 0}
     */
    public UF_HWQUPC(int n, boolean pathCompression) {
        count = n;
        parent = new int[n];
        height = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            height[i] = 1;
        }
        this.pathCompression = pathCompression;
    }

    /**
     * Initializes an empty union-find data structure with {@code n} sites
     * {@code 0} through {@code n-1}. Each site is initially in its own
     * component.
     * This data structure uses path compression
     *
     * @param n the number of sites
     * @throws IllegalArgumentException if {@code n < 0}
```

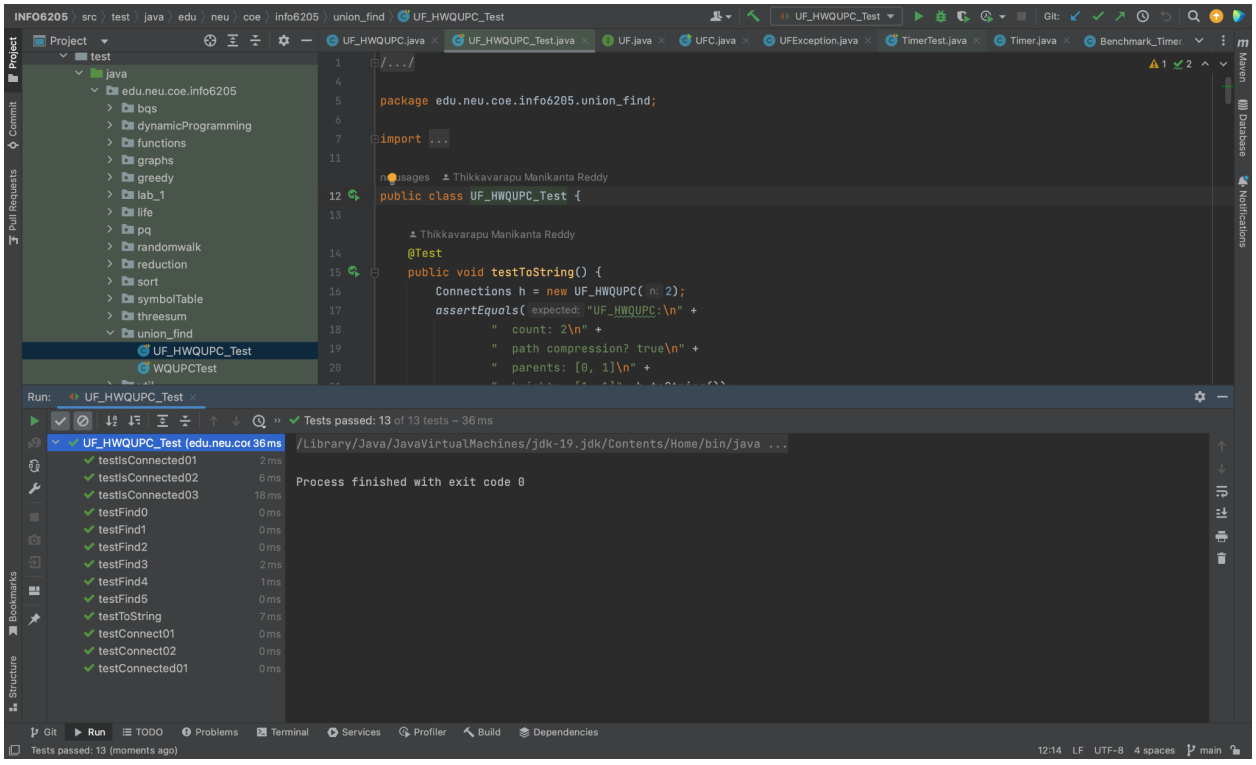
```java
     */
    public UF_HWQUPC(int n) {
        this(n, true);
    }

    public void show() {
        for (int i = 0; i < parent.length; i++) {
            System.out.printf("%d: %d, %d\n", i, parent[i], height[i]);
        }
    }

    /**
     * Returns the number of components.
     *
     * @return the number of components (between {@code 1} and {@code n})
     */
    public int components() {
        return count;
    }

    /**
     * Returns the component identifier for the component containing site
{@code p}.
     *
     * @param p the integer representing one site
     * @return the component identifier for the component containing site
{@code p}
     * @throws IllegalArgumentException unless {@code 0 <= p < n}
     */
    public int find(int p) {
        validate(p);
        int root = p;
        // FIXME
        // END

        if(pathCompression){
            doPathCompression(p);
            root = parent[root];
        }
        else {
            while(root != parent[root]){
                root = parent[root];
            }
        }

        return root;
    }

    /**
```

```
    * Returns true if the the two sites are in the same component.
    *
    * @param p the integer representing one site
    * @param q the integer representing the other site
    * @return {@code true} if the two sites {@code p} and {@code q} are in the
same component;
    * {@code false} otherwise
    * @throws IllegalArgumentException unless
    *                                  both {@code 0 <= p < n} and {@code 0 <=
q < n}
    */
   public boolean connected(int p, int q) {
       return find(p) == find(q);
   }

   /**
    * Merges the component containing site {@code p} with the
    * the component containing site {@code q}.
    *
    * @param p the integer representing one site
    * @param q the integer representing the other site
    * @throws IllegalArgumentException unless
    *                                  both {@code 0 <= p < n} and {@code 0 <=
q < n}
    */
   public void union(int p, int q) {
       // CONSIDER can we avoid doing find again?
       mergeComponents(find(p), find(q));
       count--;
   }

   @Override
   public int size() {
       return parent.length;
   }

   /**
    * Used only by testing code
    *
    * @param pathCompression true if you want path compression
    */
   public void setPathCompression(boolean pathCompression) {
       this.pathCompression = pathCompression;
   }

   @Override
   public String toString() {
       return "UF_HWQUPC:" + "\n  count: " + count +
               "\n  path compression? " + pathCompression +
```

```
                "\n  parents: " + Arrays.toString(parent) +
                "\n  heights: " + Arrays.toString(height);
    }

    // validate that p is a valid index
    private void validate(int p) {
        int n = parent.length;
        if (p < 0 || p >= n) {
            throw new IllegalArgumentException("index " + p + " is not between 0
and " + (n - 1));
        }
    }

    private void updateParent(int p, int x) {
        parent[p] = x;
    }

    private void updateHeight(int p, int x) {
        height[p] += height[x];
    }

    /**
     * Used only by testing code
     *
     * @param i the component
     * @return the parent of the component
     */
    private int getParent(int i) {
        return parent[i];
    }

    private final int[] parent;   // parent[i] = parent of i
    private final int[] height;   // height[i] = height of subtree rooted at i
    private int count;  // number of components
    private boolean pathCompression;

    private void mergeComponents(int i, int j) {
        // FIXME make shorter root point to taller one
        // END

        if (i==j) return;
        else if (height[i] == height[j]) {
            parent[j]=i;
            height[i]++;
        }
        else if(height[i] < height[j]) {
            parent[i] = j;
        }
        else {
```

```
            parent[j] = 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

        while(i != parent[i]){
            parent[i] = parent[parent[i]];
            i = parent[i];
        }

    }
}
```

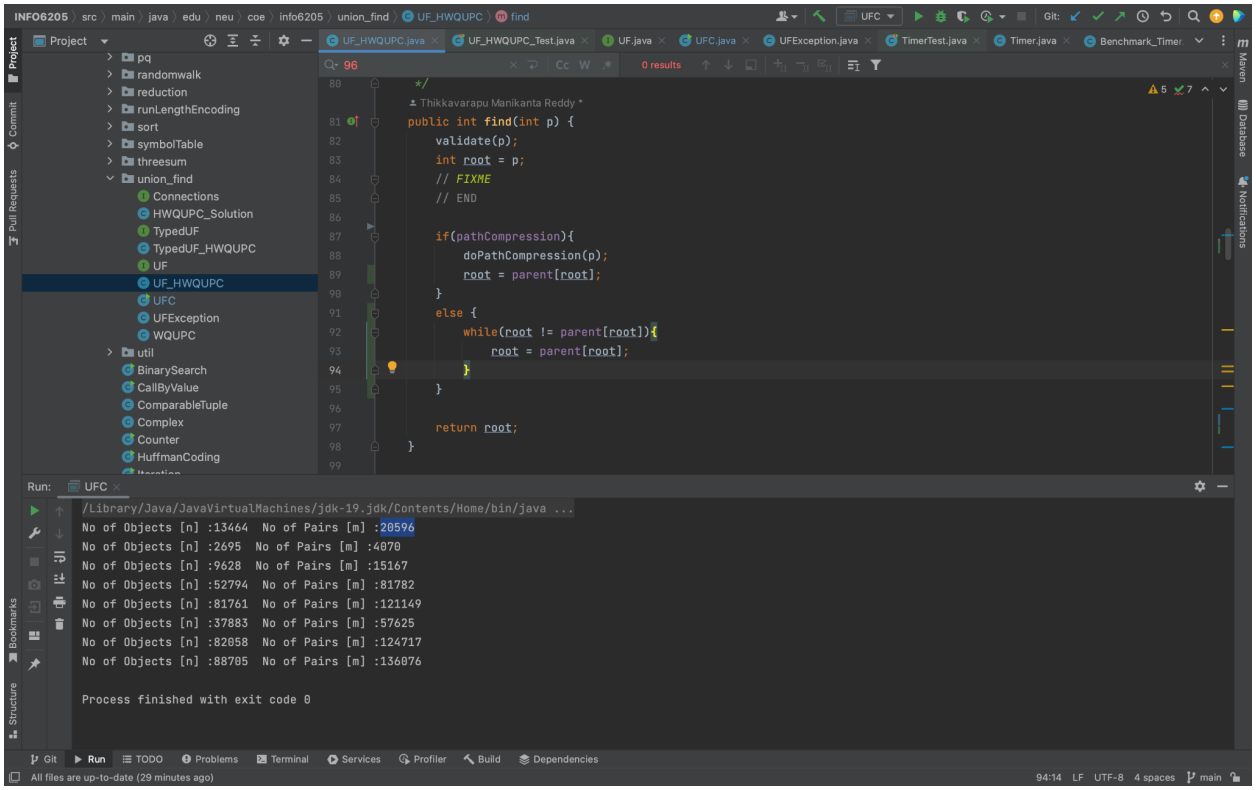
Unit Test Cases :



UF Client code changes :

```java
package edu.neu.coe.info6205.union_find;

import java.util.Random;

public class UFC {

    public static int count(int n) {
        int noc = 0; Random rc = new Random();
        UF_HWQUPC ufc = new UF_HWQUPC(n, true);
        while (ufc.components()!=1)
        {
            int p = rc.nextInt(n);
            int q = rc.nextInt(n);
            noc++;
            if(!ufc.connected(p,q)) {
                ufc.union(p, q);
            }
        }
        return noc;
    }

    public static void main(String[] args) {
        Random rand = new Random();
        for(int i=0; i<8;i++) {
            int n = rand.nextInt(99999);
            System.out.println("No of Objects [n] :" + n + "  No of Pairs [m] :"
+ count(n));
        }
    }

}
```

Output :



Observation:

It is observed that as the value of No of Objects (n) increases No of Pairs(m) value also increases and m is 1.5 times of n [m = 1.5n]. So, m is directly proportional to n.