Program Structures & Algorithms
Spring 2022
Assignment No.3

Name: Roja Pinnamraju
(NUID):002925814

Task:

1.a) Implement height-weighted Quick Union with Path Compression.

I have implemented methods as follows:

a) doPathCompression(int i): updated parent to value of grandparent.

b) find(int p): return the root of p.

c) mergeComponents(int i , int j): added logic .

2. Develop HWQUPC_Solution

Implemented HWQUPC_Solution.java that generates a number of connections with a number of objects. I have created a main method that takes random values, calls count() method, and prints the number of connections.

OUTPUT :

1)

```
/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java ...
244 744
561 1931
251 771
842 3085
980 3665
776 2815
571 1985
360 1167
339 1089
287 895
999 3741
205 607
887 3272
587 2034
615 2159
883 3251
222 667
587 2031
264 816
530 1819
892 3286
921 3423
930 3451
514 1760
399 1313
379 1243
419 1391
895 3316
672 2382
889 3283

Process finished with exit code 0
```

2)

```
/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java ...
465 1566
384 1259
673 2390
964 3581
902 3341
620 2179
293 920
622 2181
573 1978
536 1843
840 3082
679 2421
531 1819
615 2152
654 2318
520 1777
964 3599
540 1858
457 1535
644 2266
858 3149
313 991
477 1600
540 1855
727 2604
291 912
275 853
945 3510
828 3020
513 1756

Process finished with exit code 0
```

Relationship Conclusion :

I conducted several runs for different n values to check the relation. In all the runs I could see there was an increase in the number of pairs required as the n value increases. The number of pairs formed(m) increases vastly as the number of objects(n) increases.

- On plotting the graph, I could see there is almost a linearithmic relationship between a number of objects and connecting pairs.

- Time taken for components to reduce 1 will depend on the number of objects taken.

The relationship of the number of pairs needed to reduce components from n objects to 1 would be

*m = f(n) = 0.5 x n * ln(n)*

Evidence/Graph :

For larger values of n, although not equal, the average number of pairs needed to reduce the components to 1 is close to *0.5 x n * ln(n).*

In this union-find operation, we check if the pairs are connected or disconnected (n ln(n)). There are only two possibilities for each pair. Hence, the relationship between m and n is almost identical to *0.5 x n * ln(n).*

Below are the results for the performed simulations:

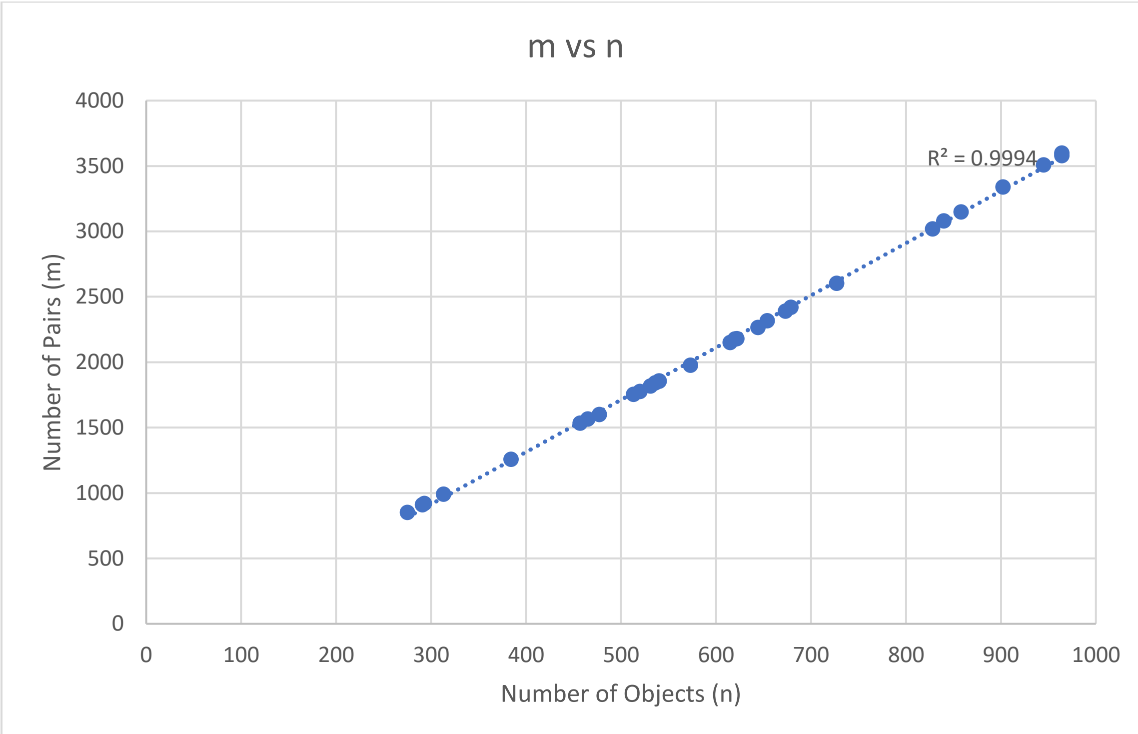
n	0.5*nlogn	m
275	772	853
291	825	912
293	832	920
313	899	991
384	1143	1259
457	1399	1535
465	1428	1566
477	1471	1600
513	1601	1756
520	1626	1777
531	1666	1819
536	1684	1843
540	1699	1858
540	1699	1855

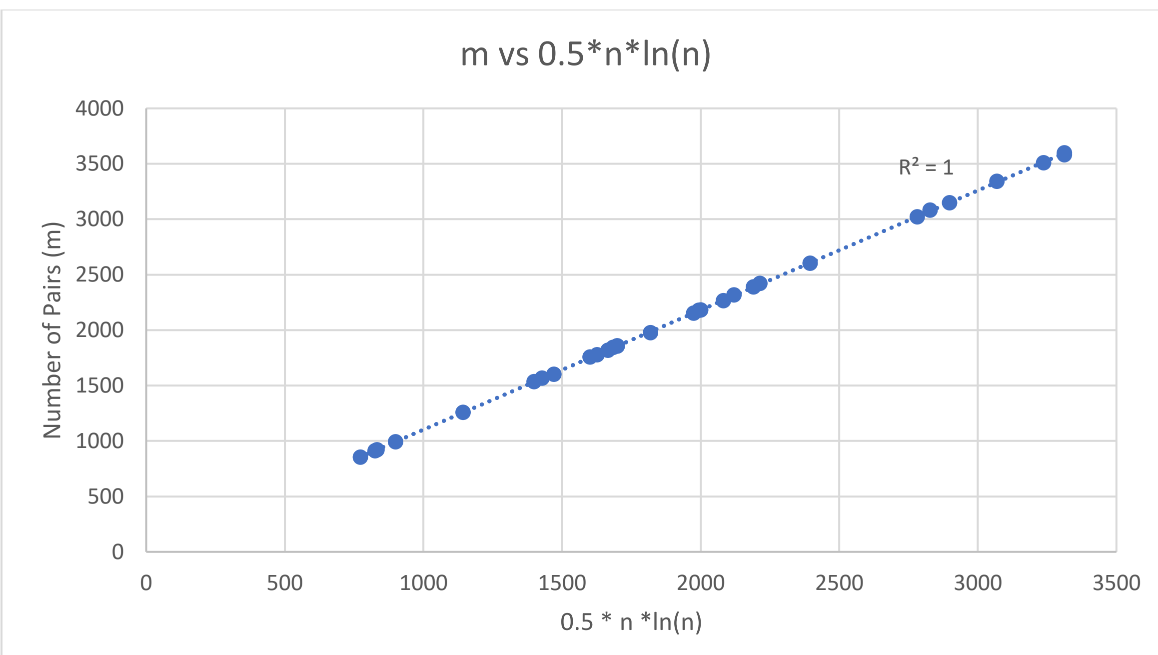
573	1820	1978
615	1975	2152
620	1993	2179
622	2001	2181
644	2083	2266
654	2120	2318
673	2191	2390
679	2214	2421
727	2395	2604
828	2782	3020
840	2828	3082
858	2898	3149
902	3069	3341
945	3237	3510
964	3312	3581
964	3312	3599

I have checked two plots to test the relationship between "n" and "m". They are as follows

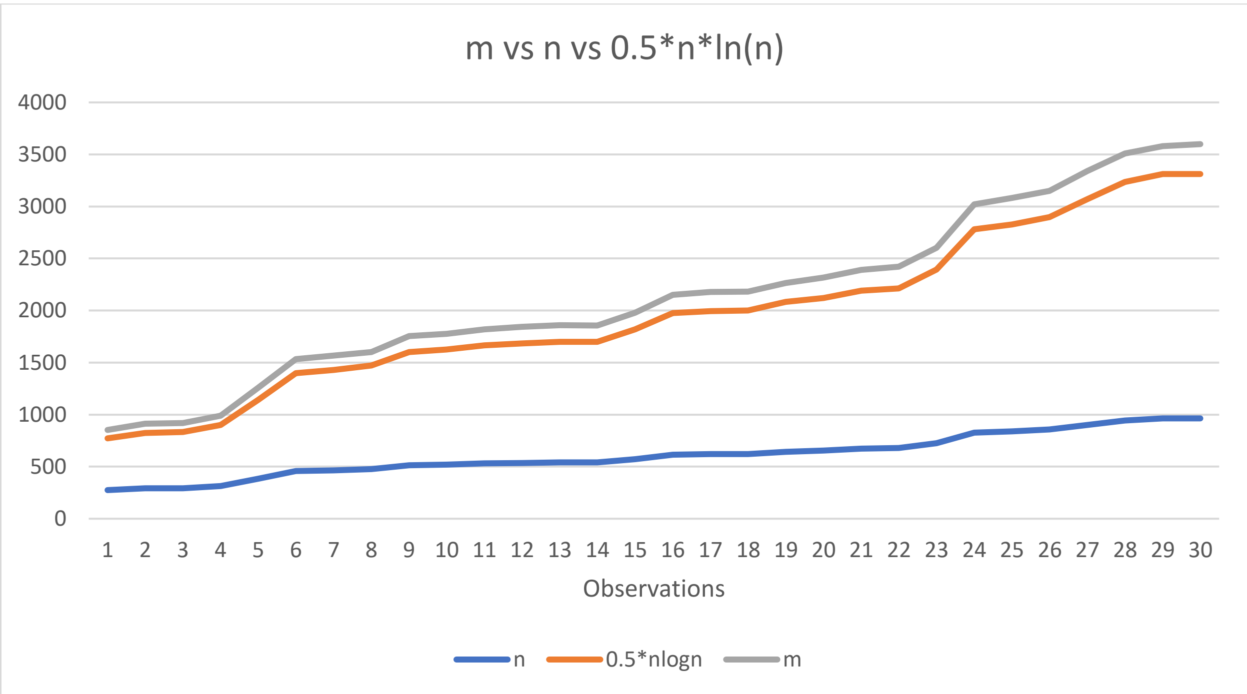
1) m vs n
2) m vs 0.5*n*ln(n)

Coefficient of determination (R^2) has been leveraged to identify the best fit among the below plots. But turns out that both the plots have similar R^2 value.


m vs n
R² = 0.9994
Number of Pairs (m)
Number of Objects (n)
4000
3500
3000
2500
2000
1500
1000
500
0
0
100
200
300
400
500
600
700
800
900
1000


m vs 0.5*n*ln(n)
R² = 1
Number of Pairs (m)
0.5 * n *ln(n)
4000
3500
3000
2500
2000
1500
1000
500
0
0
500
1000
1500
2000
2500
3000
3500

As R^2 value is not helping much here, I have plotted all the three parameters (m, n, 0.5*n*ln(n)) in a single plot for various observation points. From the plot below, it is clearly evident that "m" and "0.5*n*ln(n)* are strongly correlated and would be the best fit for our data points.



Code:

```
package edu.neu.coe.info6205.union_find;
import java.util.*;

public class HWQUPC_Solution {

    public static void main(String[] args)
    {
        int[] testdata=new int[30];int out=0;
        Random random = new Random();
        for(int i=0; i<testdata.length;i++)
testdata[i]=random.ints(200,1000).findFirst().getAsInt();
        for(int i=0;i<testdata.length;i++) {
            out=0;
```

```
            for (int j = 0; j < 5000; j++) {
                out += count(testdata[i]);

            }

            System.out.println(testdata[i]+" "+out / 5000);
           // System.out.println("For "+out+" objects, number of connections
="+out);
        }
        }

    public static int count(int i)
    {
        int randoms=0;
        UF_HWQUPC uf=new UF_HWQUPC(i,true);
        Random random= new Random();
        while(uf.components()>1)
        {
            int a= random.ints(0,i).findFirst().getAsInt();
            int b= random.ints(0,i).findFirst().getAsInt();
            randoms++;
            if(!uf.isConnected(a,b)){
                uf.union(a,b);
            }
        }
        return randoms;
    }
}
```

```
/**
 * 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,
```

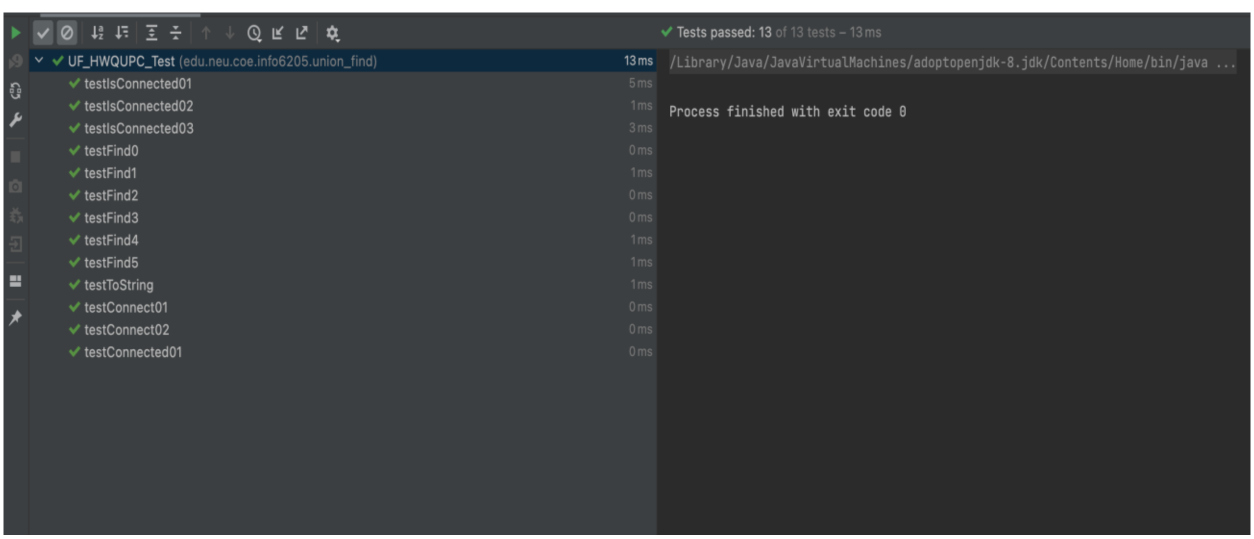
```java
     *
     * @param p the integer representing one site
     * @param q the integer representing the other site
     */
    public void connect(int p, int q) {
        if (!isConnected(p, q)) union(p, q);
    }

    /**
     * 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}
     */
    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;
    }
```

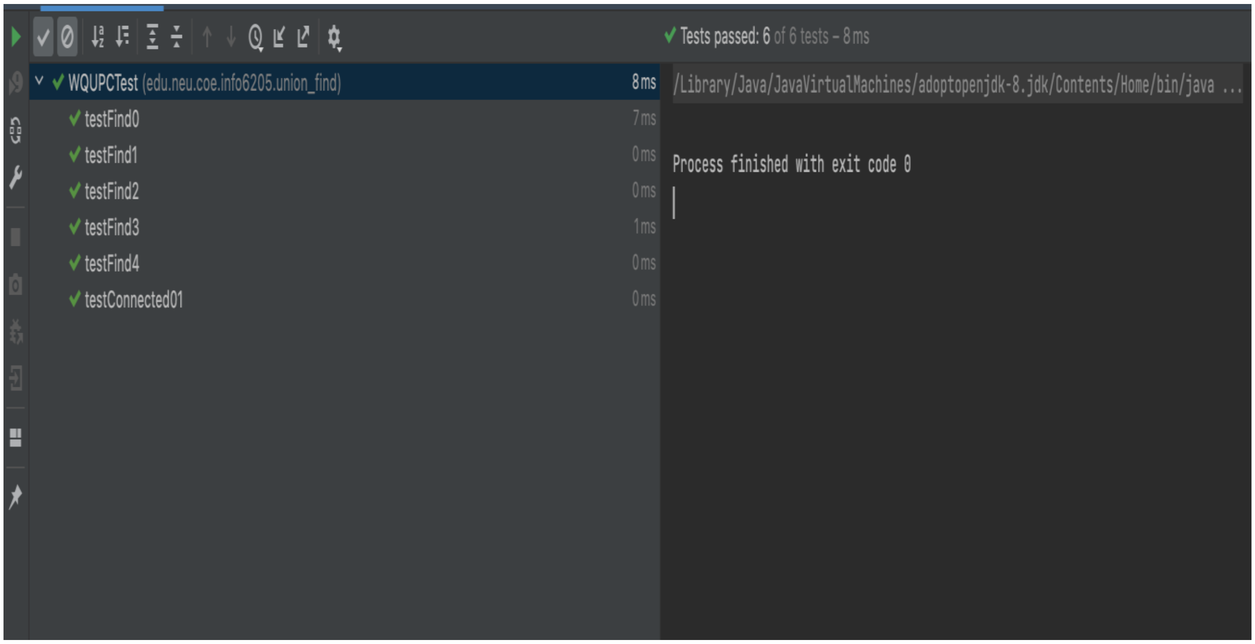
```java
    /**
     * Returns the component identifier for the component containing site
{@code p}.
     *
     * @param p the integer representing one site
     * @return the component identifier for the component containing site
{@code p}
     * @throws IllegalArgumentException unless {@code 0 <=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

        while(root!=parent[root]){

            if(pathCompression)doPathCompression(root);

            root=parent[root];

        }
        // FIXME
        // END
        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--;
    }
```

```java
    @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;
```

```java
    private void mergeComponents(int i, int j) {
        // FIXME make shorter root point to taller one
        if(height[i]<height[j]) {

            updateParent(i,j);

            updateHeight(j, i);

        }

        else {

            updateParent(j,i);

            updateHeight(i, j);

        }

    }


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

Unit test Results:




Tests passed: 6 of 6 tests – 8 ms
WQUPCTest (edu.neu.coe.info6205.union_find)
8 ms
testFind0
7 ms
testFind1
0 ms
testFind2
0 ms
testFind3
1 ms
testFind4
0 ms
testConnected01
0 ms
/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java ...
Process finished with exit code 0