Assignment 4

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).

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).

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.

Answer

Step 1

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

import java.util.Arrays;

/**
 * Height-weighted Quick Union with Path Compression
 */
public class UF_HWQUPC implements UF {
```

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

public void show() {
```

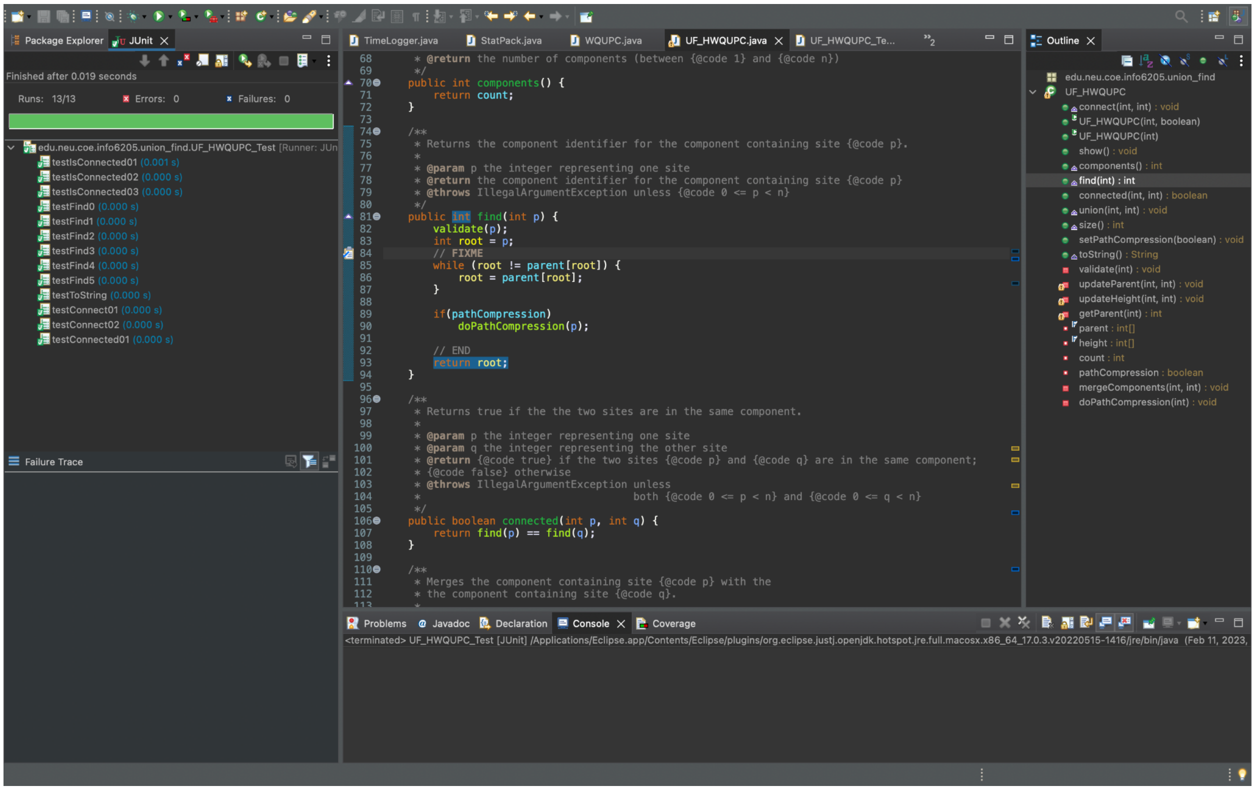
```
        for (int i = 0; i < parent.length; i++) {
            System.out.printf("%d: %d, %d\n", i, parent[i], height[i]);
        }
    }

    /**
     * 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
        while (root != parent[root]) {
            root = parent[root];
        }

        if(pathCompression)
            doPathCompression(p);

        // 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
```

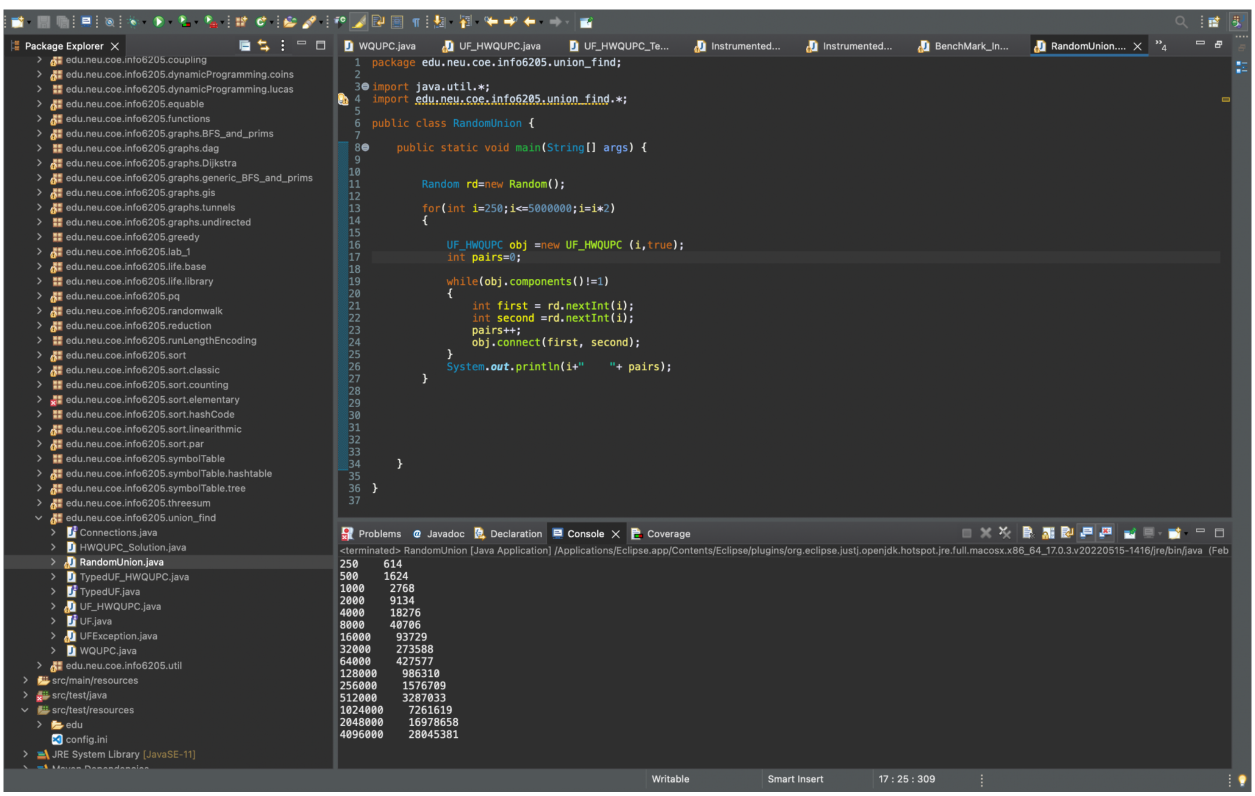
```
 * @throws IllegalArgumentException unless
 *                              both {@code 0 <= p < n} and {@code 0 <= q < n}
 */
public boolean connected(int p, int q) {
    return find(p) == find(q);
}

/**
 * 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--;
}

@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);
}
```

```java
// 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
        int rootP = find(i);
    int rootQ = find(j);

    if (rootP == rootQ) return;
    // make smaller root point to larger one
    if (height[rootP] < height[rootQ]) {
        parent[rootP] = rootQ;
        height[rootQ] += height[rootP];
    } else {
        parent[rootQ] = rootP;
        height[rootP] += height[rootQ];
```

```
        }
          // END
    }

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

B. The unit test cases

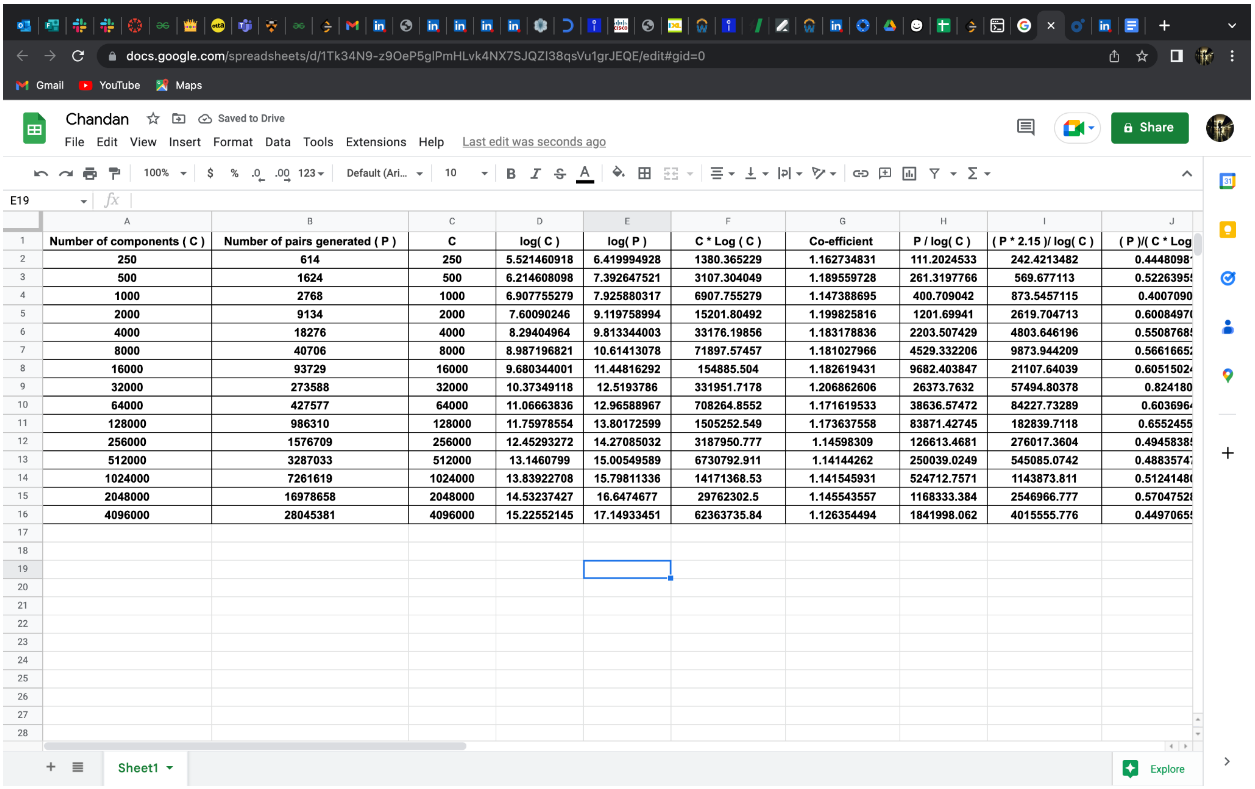

Package Explorer
JUnit
Finished after 0.019 seconds
Runs: 13/13
Errors: 0
Failures: 0
edu.neu.coe.info6205.union_find.UF_HWQUPC_Test [Runner: JUn
testIsConnected01 (0.001 s)
testIsConnected02 (0.000 s)
testIsConnected03 (0.000 s)
testFind0 (0.000 s)
testFind1 (0.000 s)
testFind2 (0.000 s)
testFind3 (0.000 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)
Failure Trace
TimeLogger.java
StatPack.java
WQUPC.java
UF_HWQUPC.java
UF_HWQUPC_Te...
* @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
while (root != parent[root]) {
root = parent[root];
}
if(pathCompression)
doPathCompression(p);
// 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}.
Problems
Javadoc
Declaration
Console
Coverage
<terminated> UF_HWQUPC_Test [JUnit] /Applications/Eclipse.app/Contents/Eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.3.v20220515-1416/jre/bin/java (Feb 11, 2023,
Outline
edu.neu.coe.info6205.union_find
UF_HWQUPC
connect(int, int) : void
UF_HWQUPC(int, boolean)
UF_HWQUPC(int)
show() : void
components() : int
find(int) : int
connected(int, int) : boolean
union(int, int) : void
size() : int
setPathCompression(boolean) : void
toString() : String
validate(int) : void
updateParent(int, int) : void
updateHeight(int, int) : void
getParent(int) : int
parent : int[]
height : int[]
count : int
pathCompression : boolean
mergeComponents(int, int) : void
doPathCompression(int) : void


Package Explorer
WQUPC.java
UF_HWQUPC.java
UF_HWQUPC_Te...
Instrumented...
Instrumented...
BenchMark_In...
RandomUnion....
package edu.neu.coe.info6205.union_find;
import java.util.*;
import edu.neu.coe.info6205.union_find.*;
public class RandomUnion {
public static void main(String[] args) {
Random rd=new Random();
for(int i=250;i<=5000000;i=i*2)
{
UF_HWQUPC obj =new UF_HWQUPC (i,true);
int pairs=0;
while(obj.components()!=1)
{
int first = rd.nextInt(i);
int second =rd.nextInt(i);
pairs++;
obj.connect(first, second);
}
System.out.println(i+"   "+ pairs);
}
}
}
Problems
Javadoc
Declaration
Console
Coverage
<terminated> RandomUnion [Java Application] /Applications/Eclipse.app/Contents/Eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.3.v20220515-1416/jre/bin/java (Feb
250 614
500 1624
1000 2768
2000 9134
4000 18276
8000 40706
16000 93729
32000 273588
64000 427577
128000 986310
256000 1576709
512000 3287033
1024000 7261619
2048000 16978658
4096000 28045381
Writable
Smart Insert
17 : 25 : 309

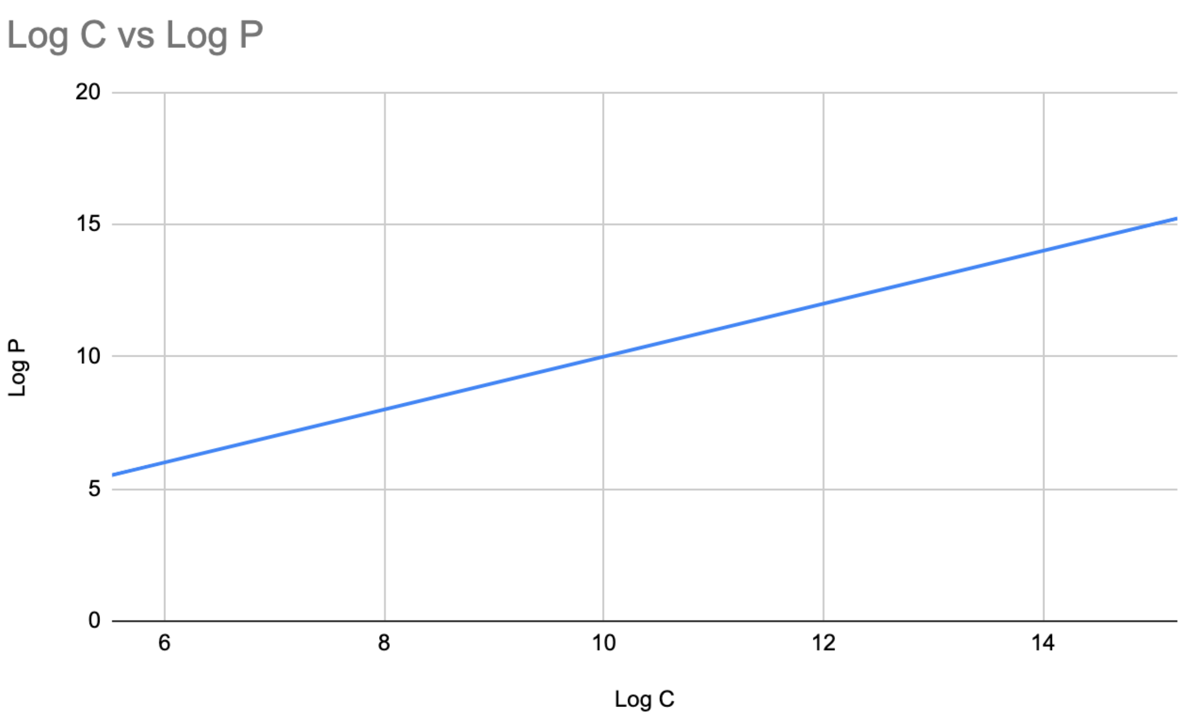
Step 2:

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

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

public class RandomUnion {

	public static void main(String[] args) {


		Random rd=new Random();

		for(int i=250;i<=5000000;i=i*2)
		{

			UF_HWQUPC obj =new UF_HWQUPC (i,true);
			int pairs=0;

			while(obj.components()!=1)
			{
				int first = rd.nextInt(i);
				int second =rd.nextInt(i);
				pairs++;
				obj.connect(first, second);
			}
			System.out.println(i+"    "+ pairs);
		}

	}
```

Step 3:



Number of components (C)	Number of pairs generated (P)	C	log(C)	log(P)	C * Log (C)	Co-efficient	P / log(C)	(P * 2.15)/ log(C)	(P)/(C * Log
250	614	250	5.521460918	6.419994928	1380.365229	1.162734831	111.2024533	242.4213482	0.4448098
500	1624	500	6.214608098	7.392647521	3107.304049	1.189559728	261.3197766	569.677113	0.5226395
1000	2768	1000	6.907755279	7.925880317	6907.755279	1.147388695	400.709042	873.5457115	0.4007090
2000	9134	2000	7.60090246	9.119758994	15201.80492	1.199825816	1201.69941	2619.704713	0.6008497
4000	18276	4000	8.29404964	9.813344003	33176.19856	1.183178836	2203.507429	4803.646196	0.5508768
8000	40706	8000	8.987196821	10.61413078	71897.57457	1.181027966	4529.332206	9873.944209	0.5661665
16000	93729	16000	9.680344001	11.44816292	154885.504	1.182619431	9682.403847	21107.64039	0.6051502
32000	273588	32000	10.37349118	12.5193786	331951.7178	1.206862606	26373.7632	57494.80378	0.824180
64000	427577	64000	11.06663836	12.96588967	708264.8552	1.171619533	38636.57472	84227.73289	0.603696
128000	986310	128000	11.75978554	13.80172599	1505252.549	1.173637558	83871.42745	182839.7118	0.6552455
256000	1576709	256000	12.45293272	14.27085032	3187950.777	1.14598309	126613.4681	276017.3604	0.4945838
512000	3287033	512000	13.1460799	15.00549589	6730792.911	1.14144262	250039.0249	545085.0742	0.4883574
1024000	7261619	1024000	13.83922708	15.79811336	14171368.53	1.141545931	524712.7571	1143873.811	0.5124148
2048000	16978658	2048000	14.53237427	16.6474677	29762302.5	1.145543557	1168333.384	2546966.777	0.5704752
4096000	28045381	4096000	15.22552145	17.14933451	62363735.84	1.126354494	1841998.062	4015555.776	0.4497065

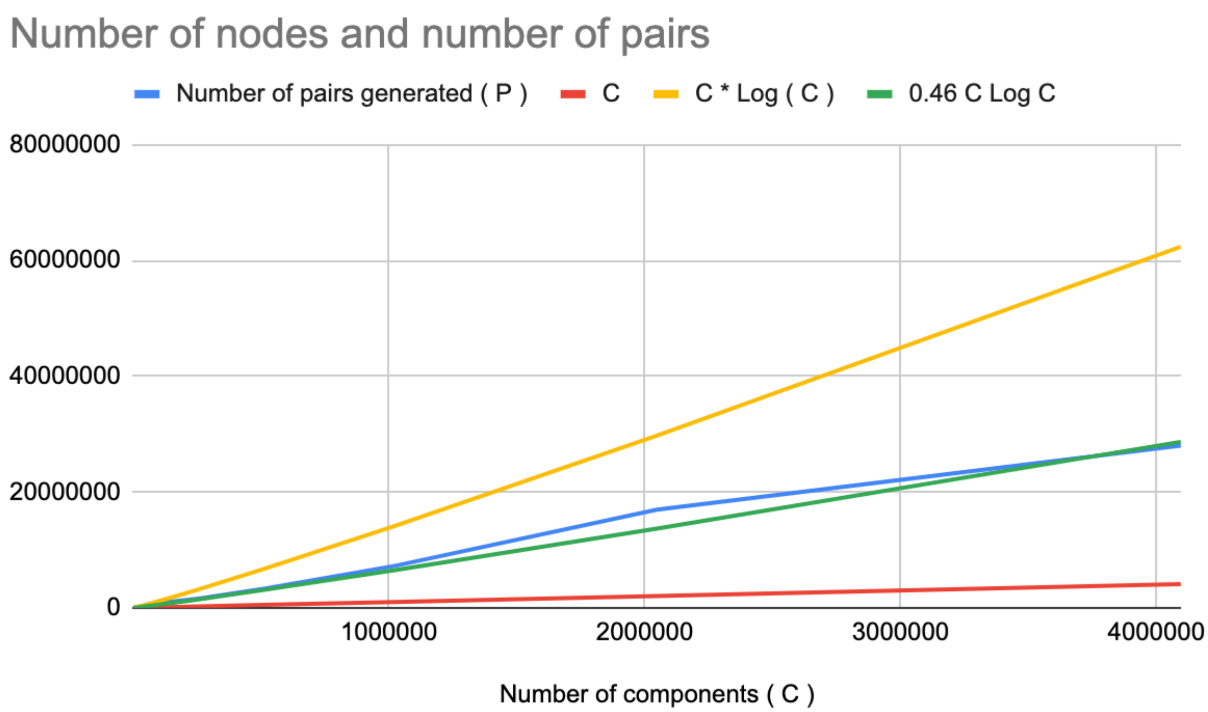


From the Log p and log c graph we can say that the coefficient is not quadratic and it has a co efficient equal to 1.12 which says that it is not linear also

The above equation has a co efficient value greater than 1 so this is not linear.

Consider the value of P/log(C) where we get the value which are still greater than 1 so after dividing the value by P/(C * log C) then we have a constant value which is equal to 0.46 which is a constant is almost equal to half and hence the relation ship between P and C are it has a constant value as .46 * C * Log(C)

P is approximately equal to N log N with a constant value getting multiplied.



From the above graph the value of C Vs P is linear and and the value of C * Log C is too high for the value of P and therefore the relationship between the C and P are Constant * C *log C

Therefore

P=0.46*C*Log C