Consider that numAtoms is n,

```
public static double calc(double[][] coords, int numAtoms) {

    double energySum = 0.0;
    double r0 = 1.2;

    for (int i=0; i < numAtoms-1; i++) {
        for (int j= i+1; j < numAtoms; j++) {
            double distance = Math.sqrt( (coords[i][0] - coords[j][0]) * (coords[i][0] - coords[j][0]) +
                                         (coords[i][1] - coords[j][1]) * (coords[i][1] - coords[j][1]) +
                                         (coords[i][2] - coords[j][2]) * (coords[i][2] - coords[j][2])  );

            double partialTerm1cubed= (r0/distance) * (r0/distance)*(r0/distance);
            double term2 = partialTerm1cubed * partialTerm1cubed ;
            double term1 = term2 * term2;

            energySum = energySum + (term1 - 2.0 * term2);
        }
    }
    return energySum;
}
```

This for loop starts at 'i + 1' and goes until n. This essentially leaves the same trace as the first program but divides work by half

All assignments and operators here costs 26 ticks total.

This loop runs for 'n-1' times

Due to the reasons above, the program will run $\frac{n^2-n}{2}$ times and cost $26\left(\frac{n^2-n}{2}\right)$ ticks. The numerator dividing by two makes a huge difference compared to the first algorithm, but in terms of big-O notation, the function is defined as $\frac{1}{2}(n^2-n)$ So thus, according to the Big-Oh rules, the function remains at a $O(n^2)$ complexity.