/* Saury Khanal / Havin Lim / Quynh Dao
Lab: 14 - Empirical Complexity Analysis
September 24th 2022
Sources : None
Help obtained : Kevin (Mentor) and Professor Jimenez
We confirm that the above list of sources is complete AND that we have not talked to anyone else (e.g., CSC207 students) about the solution to this problem
*/

#1.
All three compute functions run relatively fast (around 0.055-0.007) for smaller array sizes (from 10 to 500) but for larger size arrays (n >= 1000) compute 1 takes considerably longer time for it to compute the wall-clock time compared to the other two compute functions. Both compute2 and compute3 run quite fast even when the array sizes increase substantially but compute3 still seems to run faster (the runtime difference is clearer when array size >= 5000)
We think we receive different results because the complexity of the codes are all different for the three compute functions. (compute1 runs in O(N^3) because it has 3 nested for loops, compute2 runs in O(N^2) because it has 2 nested for loops, compute3 runs in O(N) because it has a single for loop)

WALL-CLOCK EXPERIMENT:

// compute1

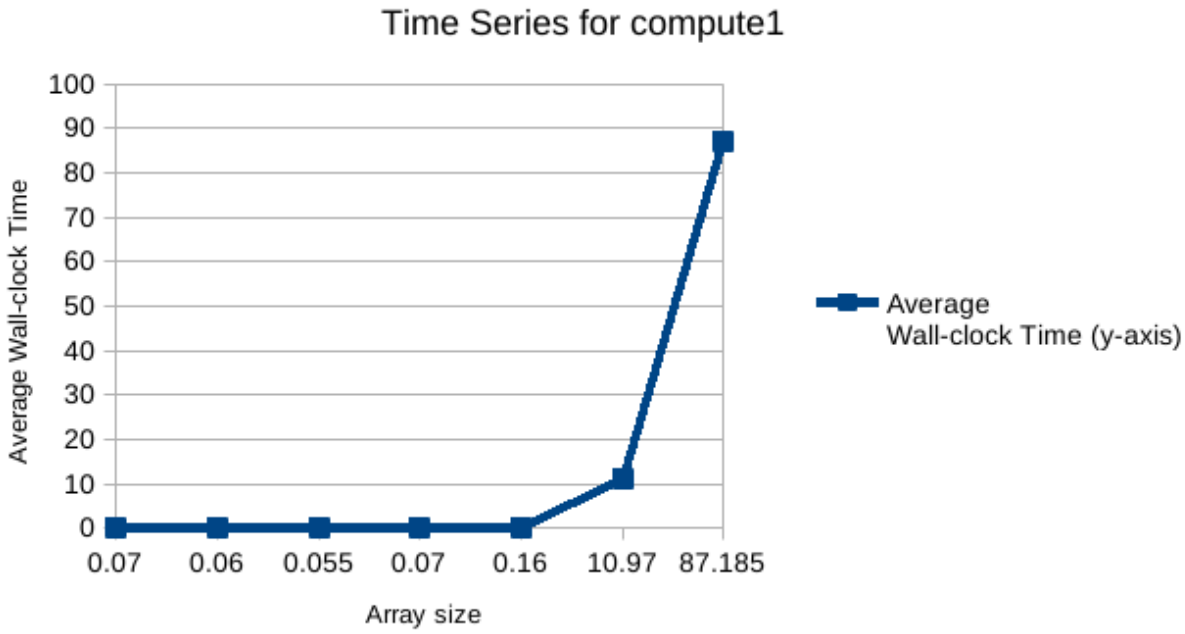
Array size (x-axis)	10	50	100	500	1000	5000	10000
Wall-clock Time	0.07	0.07	0.08	0.08	0.16	11.03	87.28
Wall-clock Time	0.08	0.07	0.07	0.08	0.18	10.96	87.17
Wall-clock Time	0.06	0.05	0.04	0.06	0.14	10.98	87.2
Wall-clock Time	0.04	0.04	0.04	0.06	0.14	10.96	87.17
Wall-clock Time	0.05	0.04	0.04	0.06	0.14	10.97	87.3
Average Wall-clock Time (y-axis)	0.07	0.06	0.055	0.07	0.16	10.97	87.185

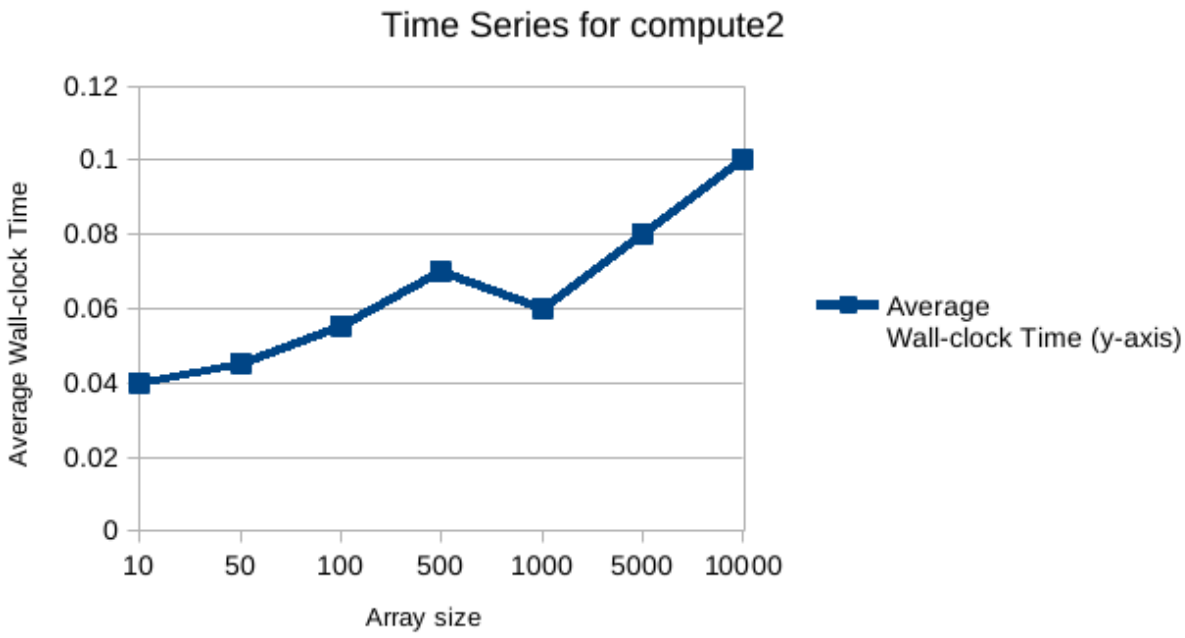
`// compute2`

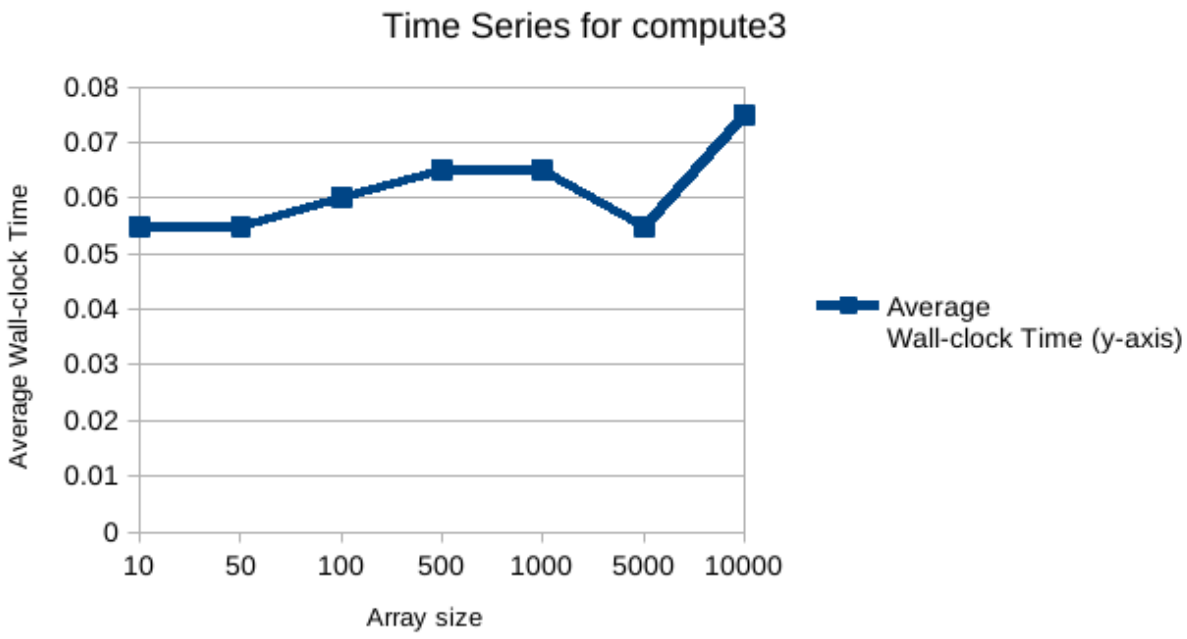
Array size (x-axis)	10	50	100	500	1000	5000	10000
Wall-clock Time	0.07	0.07	0.06	0.07	0.08	0.09	0.13
Wall-clock Time	0.04	0.05	0.06	0.08	0.06	0.09	0.11
Wall-clock Time	0.04	0.04	0.05	0.06	0.06	0.07	0.09
Wall-clock Time	0.06	0.04	0.04	0.06	0.06	0.06	0.08
Wall-clock Time	0.04	0.04	0.04	0.05	0.06	0.06	0.08
Average Wall-clock Time (y-axis)	0.04	0.045	0.055	0.07	0.06	0.08	0.1

`// compute3`

Array size (x-axis)	10	50	100	500	1000	5000	10000
Wall-clock Time	0.06	0.07	0.06	0.06	0.05	0.07	0.1
Wall-clock Time	0.07	0.04	0.05	0.06	0.06	0.06	0.09
Wall-clock Time	0.04	0.07	0.07	0.07	0.07	0.05	0.06
Wall-clock Time	0.05	0.05	0.08	0.06	0.07	0.07	0.08
Wall-clock Time	0.04	0.05	0.05	0.06	0.05	0.05	0.06
Average Wall-clock Time (y-axis)	0.055	0.055	0.06	0.065	0.065	0.055	0.075

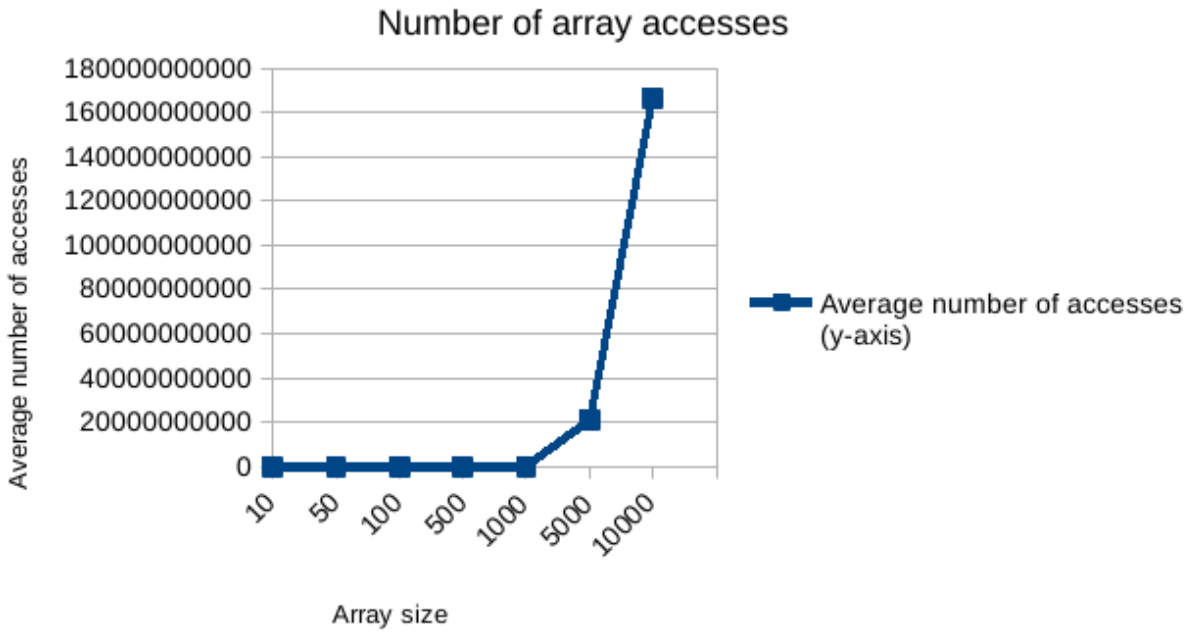

Time Series for compute1
Average Wall-clock Time
100
90
80
70
60
50
40
30
20
10
0
0.07
0.06
0.055
0.07
0.16
10.97
87.185
Array size
Average
Wall-clock Time (y-axis)


Time Series for compute2
Average Wall-clock Time
0.12
0.1
0.08
0.06
0.04
0.02
0
10
50
100
500
1000
5000
10000
Array size
Average
Wall-clock Time (y-axis)



#2.

Array size (x-axis)	10	50	100	500	1000	5000	10000
Number of array accesses	220	22100	171700	20958500	167167000	2084583500	166716670000
Number of array accesses	220	22100	171700	20958500	167167000	2084583500	166716670000
Number of array accesses	220	22100	171700	20958500	167167000	2084583500	166716670000
Number of array accesses	220	22100	171700	20958500	167167000	2084583500	166716670000
Number of array accesses	220	22100	171700	20958500	167167000	2084583500	166716670000
Average number of accesses (y-axis)	220	22100	171700	20958500	167167000	2084583500	166716670000


Number of array accesses
180000000000
160000000000
140000000000
120000000000
100000000000
80000000000
60000000000
40000000000
20000000000
0
Average number of accesses
10
50
100
500
1000
5000
10000
Array size
Average number of accesses (y-axis)

```
/*  Saury Khanal  / Havin Lim / Quynh Dao
Lab: 15 - Empirical Complexity Analysis
September 26th 2022
Sources : None
Help obtained : Kevin (Mentor) and Professor Jimenez
We confirm that the above list of sources is complete AND that we have not
talked to anyone else (e.g., CSC207 students) about the solution to this
problem
*/

Exercise 1
   (1) O(N^2)
   (2) quadratic time
   (3) n == 10 / real 0.08
       n == 100 / real 0.06
       n == 1000 / real 0.06
       n == 10000 / real 0.08

Exercise 2
   (1) O(2N)
   (2) linear time
   (3) n == 10 / real 0.05
       n == 100 / real 0.07
       n == 1000 / real 0.05
       n == 10000 / real 0.05
   (4) Our analysis is that the function runs in linear time, which means the
      runtimes will be directly proportional to N. (runtimes*10 when n*10).
      However, our complexity model does not align with the real time since
      the actual running time doesn't multiply by 10 when n multiply by 10.

Exercise 3
   (1) O(N^5)
   (2) quintic time
   (3) n == 10 / real 0.05
       n == 50 / real 0.06
       n == 100 / real 0.06
       n == 500 / real 0.34
       n == 1,000 / real 2.32
   (4) Our analysis is that the function runs in quintic time, which means the
      runtimes will be proportional to the fifth power of the input size.
      Since we see that the values of real time increases drastically as n
      increases which matches with the quintic time we have analyzed.
```

Exercise 4

(1) O(log N) Logarithmic time

(2) n == 10 / real 0.08
n == 100 / real 0.06
n == 500 / real 0.07
n == 1,000 / real 0.06
n == 10,000 / real 0.06

Since the Big-Oh running time of Loop4 is a logarithmic time the results for the time measurement should level off as the n increases as we analyzed (runtimes will only increase a little as n drastically increases).

Exercise 5

(1) O(N^4) quartic time

(2) n == 10 / real 0.06
n == 100 / real 0.07
n == 500 / real 0.32
n == 1,000 / real 1.83

(3) Our analysis is that the function runs in quartic time, which means the runtimes will be proportional to the fourth power of the input size. Since we see that the values of real time increases drastically as n increases which matches with the quarticic time we have analyzed (Not perfectly, but they do).

(4) It happens when j is the perfect multiple of i, but this does not happen often. As a result the innermost loop occurs rarely which differs our result widely from the Big-Oh prediction.

<TestAnalysis.java>

```
/*
Author: Saury Khanal  / Havin Lim / Quynh Dao
Lab: 15 - Empirical Complexity Analysis
September 26th 2022
Sources : None
Help obtained : Kevin (Mentor) and Professor Jimenez
We confirm that the above list of sources is complete AND that we have not
talked to anyone else (e.g., CSC207 students) about the solution to this
problem
*/

public class TestAnalysis {
    public static void main (String[] args) {
        int method;
        int runTime;

        //set runTime and method as parameter for the main method
        runTime = Integer.parseInt(args[1]);
        method = Integer.parseInt(args[0]);

        try {
            // run Loop1
            if (method == 1){
                Loop1.run(runTime);
            }
            // run Loop2
            else if (method == 2){
                Loop2.run(runTime);
            }
            // run Loop3
            else if (method == 3) {
                Loop3.run(runTime);
            }
            // run Loop4
            else if (method == 4) {
                Loop4.run(runTime);
            }
            // run Loop5
            else if (method == 5) {
                Loop5.run(runTime);
            }
```

```
            // handle error of invalid input
            else  {
                System.out.println("Input number has to be from 1 to 5");
                System.exit(1);
            }
        }
        // handle number format exception
        catch (NumberFormatException e){
            System.err.println(e);
        }
        // handle other exception
        catch (Exception e){
            System.err.println(e);
            System.exit(1);
        }
    }
}
```

```
/*  Saury Khanal  / Havin Lim / Quynh Dao
Lab: 16 - Population Density of Iowa Counties
September 30th 2022
Sources : None
Help obtained : Kevin (Mentor) and Professor Jimenez
We confirm that the above list of sources is complete AND that we have not
talked to anyone else (e.g., CSC207 students) about the solution to this
problem
*/
```

<SparseFinder.java>

```
package demographics;
import java.util.Scanner;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;

public class SparseFinder {
	// reads in the four-line entry for a county from input Scanner
	public static County readCounty(Scanner sc) throws IOException {
		// check if the file has next token to scan
		if (sc.hasNext()) {
			// read the next four-line input
			String name = sc.nextLine();
			int population = Integer.parseInt(sc.nextLine());
			int area = Integer.parseInt(sc.nextLine());
			// read the blank line if there exists
			if (sc.hasNext()) {
				sc.nextLine();
			}
			County county = new County (name, population,area);
			return county;
		}
		return null;
	}
```

```
	// return the county with least population density
	public static County findSparsest(ArrayList<County> countyList) {
		Iterator<County> c = countyList.iterator();
		// check if the file has next token to scan
		if (c.hasNext()) {
			County min = c.next();
			// while loop if the file has next token to scan
			while (c.hasNext()) {
				County current = c.next();
				// check if the current min population is larger than
the current county's population density
				if (min.populationDensity() >
current.populationDensity()) {
					min = current;
				}
			}
			// return the county with least population density
			return min;
		}
		// if there is no county, return null
		return null;
	}

	public static void main(String[] args) throws FileNotFoundException {
		ArrayList<County> counties = new ArrayList<>();
		Scanner countyDataSource = null;

		try {
			// add county to an ArrayList
			countyDataSource = new Scanner(new FileReader(args[0]));
			while (countyDataSource.hasNext()) {
				counties.add(readCounty(countyDataSource));
			}
		}
		// catch NumberFormatException
		catch(NumberFormatException e) {
			System.err.println(e);
		}
		// catch IOException
		catch (IOException ioe) {
			System.err.println(ioe);
		}
```

```
            // finally if the file is not null
            finally {
                  if (countyDataSource != null) {
                        countyDataSource.close();
                  }
            }
            // get the county with the smallest population density
            County min = findSparsest(counties);
            // print the name and population density of that county
            System.out.println("County name: " + min.getName() + "\nPopulation
density: " + min.populationDensity());
      }
}
```

<County.java>

```
package demographics;

public class County {
      private String name;
      private int population;
      private int area;

      // Constructor
      public County(String name,int population,int area) {
            this.name=name;
            this.population=population;
            this.area=area;
      }
      // method to get the name of county
      public String getName() {
            return name;
      }
      // method to get the population of county
      public int getPopulation() {
            return population;
      }
      // method to get the area of county
      public int getArea() {
            return area;
```

```
	}
	// method to get the population density of county
	public float populationDensity() {
		return (float) population/ (float) area;
	}
}
```