Patrick Phillips
Class ID : 107

Task 1: Give Big-Oh runtimes of each program

1.) Runtime is O(n^2) because the double for loop is n*n

```
public static int count(int[] a) { // O(n^2)
    int n = a.length;
    int count = 0;
    for (int i = 0; i < n; i++) { //n
        for (int j = i+1; j < n; j++) { //n
            if (a[i] + a[j] == 0) {
                count++;
            }
        }
    }
    return count;
}
```

2.) Runtime is O(nlog(n)) because the for loop is n time, and the nested binary search is log n time.

```
public static int count(int[] a) { //O(nlogn)
    int n = a.length;
    Arrays.sort(a);
    if (containsDuplicates(a)) throw new IllegalArgumentException("array contains duplicate integers");
    int count = 0;
    for (int i = 0; i < n; i++) { //n time
        int j = Arrays.binarySearch(a, -a[i]); //log n time
        if (j > i) count++;
    }
    return count;
}
```

3.)This has runtime O(n^3) because the nested for loops are n*n*n

```
public static int count(int[] a) {
    int n = a.length;
    int count = 0;
    for (int i = 0; i < n; i++) { // n time
        for (int j = i+1; j < n; j++) { // n time
            for (int k = j+1; k < n; k++) { // n time
                if (a[i] + a[j] + a[k] == 0) {
                    count++;
                }
            }
        }
    }
    return count;
}
```

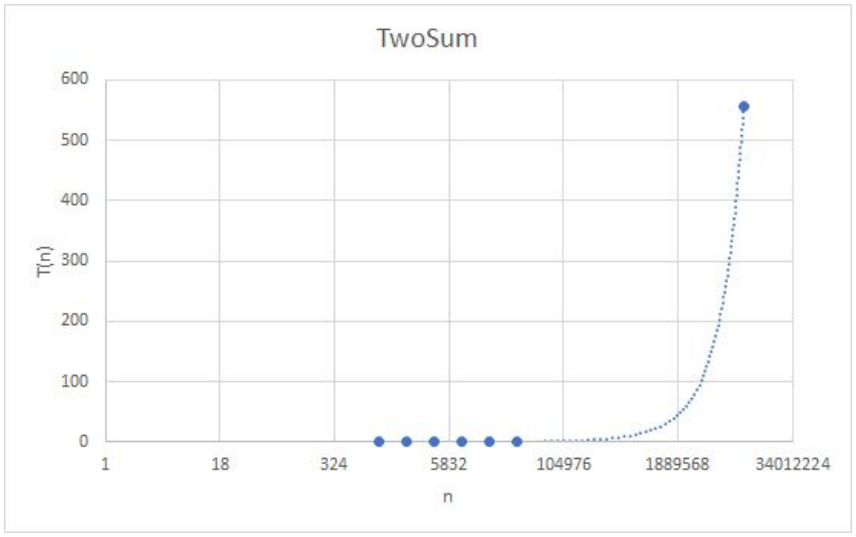
4.) This has runtime O(n^2 log n) because the binary search nested in for loop, nested in a for loop is n*n*log n

```
public static void printAll(int[] a) {
    int n = a.length;
    Arrays.sort(a);
    if (containsDuplicates(a)) throw new IllegalArgumentException("array contains duplicate integers");
    for (int i = 0; i < n; i++) { // n time
        for (int j = i+1; j < n; j++) { // n time
            int k = Arrays.binarySearch(a, -(a[i] + a[j])); //log n time
            if (k > j) StdOut.println(a[i] + " " + a[j] + " " + a[k]);
        }
    }
}
```

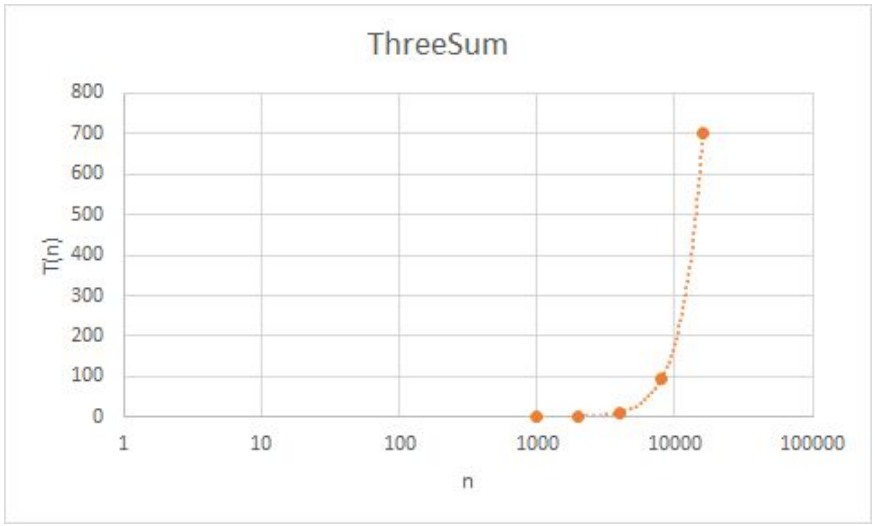
Task 2: Run each program for each integer file, provide screenshots, and graph the values using a logarithmic scale for the x-axis.

(All Screenshots are in in folder labeled “screenshots”)

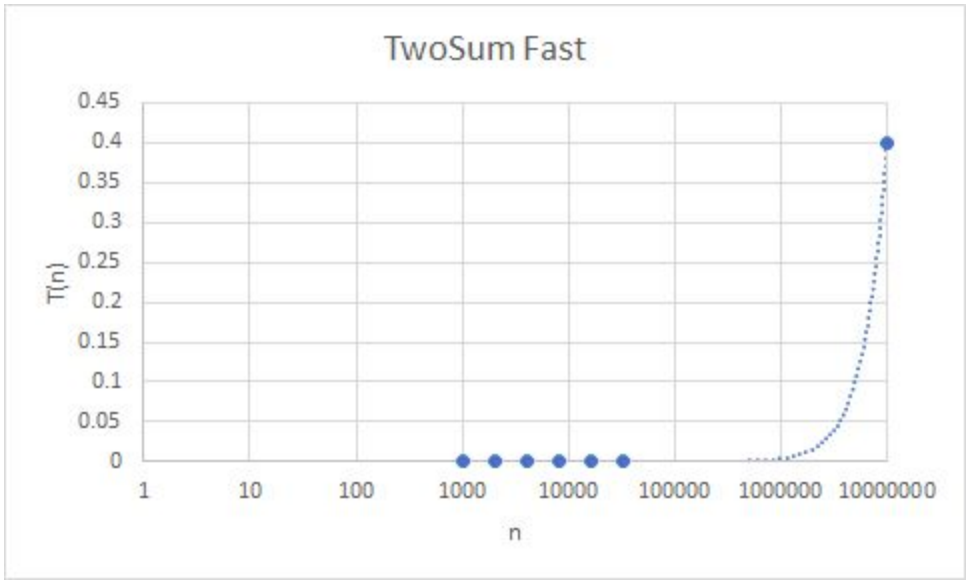
1.)



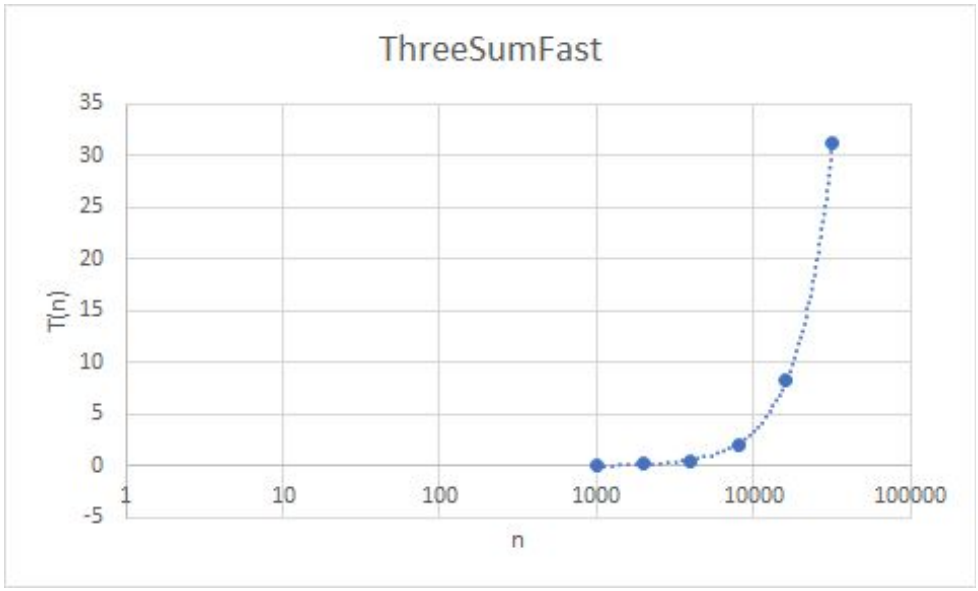
2.)



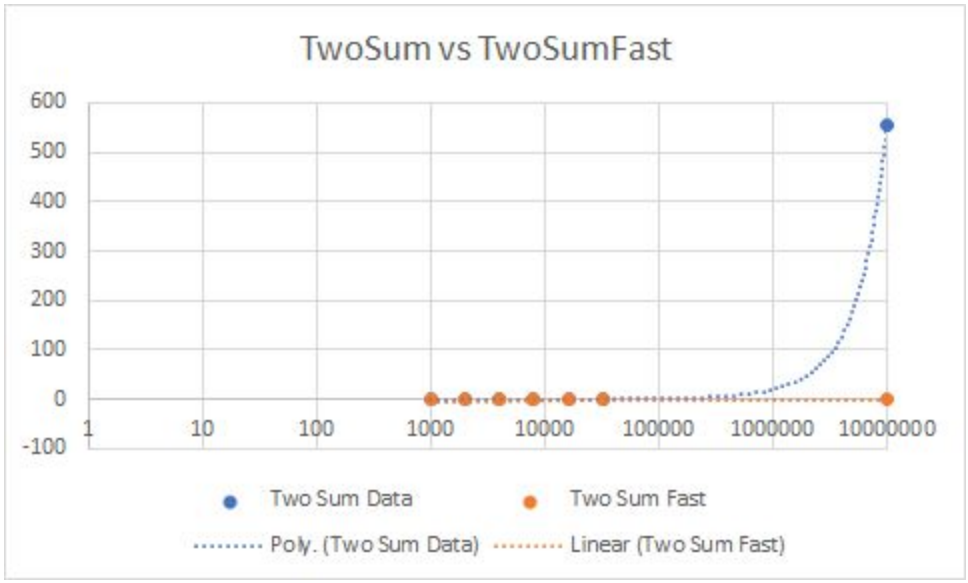
3.)


TwoSum Fast
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
T(n)
1
10
100
1000
10000
100000
1000000
10000000
n

4.)

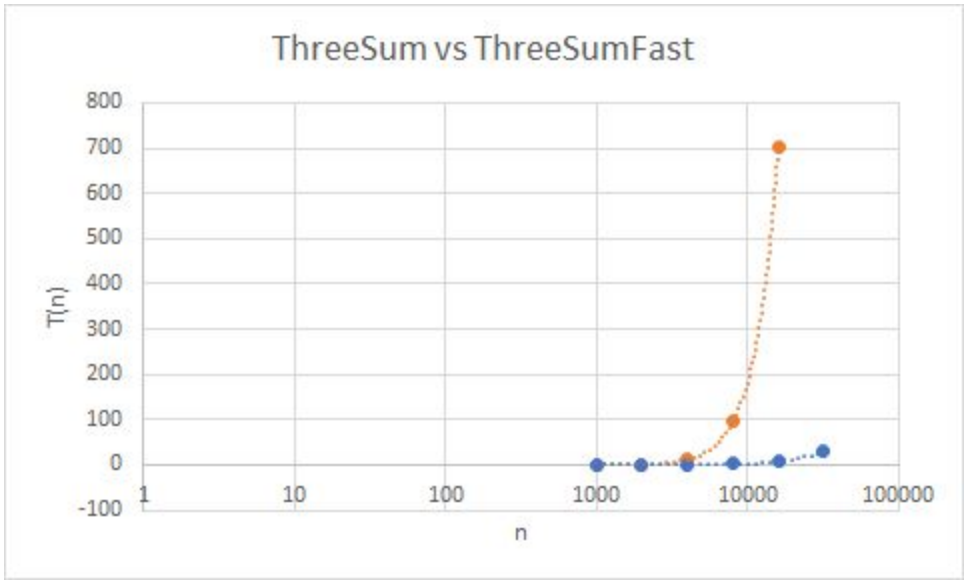

ThreeSumFast
35
30
25
20
15
10
5
0
-5
T(n)
1
10
100
1000
10000
100000
n

5.)



TwoSumFast has a much faster run time with large values of “n” inputs, and is O(TwoSum),

6.)



ThreeSumFast has a much faster run time than threesum with large amounts of input, and is O(ThreeSum).

Task 3: Compare runtimes for each pair of the consecutive run for each program, then estimate runtimes for 32K and 1M integers.

Runtimes of each program

(n) integer inputs	Two Sum	Two Sum Fast	Three Sum	ThreeSum Fast
1000	**0.0**	**0.0**	**0.2**	**0.0**
2000	**0.0**	**0.0**	**1.4**	**0.2**
4000	**0.08**	**.003**	**11.9**	**0.5**
8000	**0.12**	**.007**	**95.4**	**2.0**
16000	**0.3**	**.02**	**702.9**	**8.2**

1.) TwoSum.java estimates:

4K to 8K
.12/.08 = 1.5 times larger
8K to 16K
.3/.12 = 2.5 times larger

16K to 32K
t(n) / .3 = 2 times larger *(estimate based on growth rate of~ 2)*
t(n) = 1.3 for 32K ints

32K to 1M
31.25~(2^5) (*doubles about five times)*
t(n)/1.3 = 4.2 * (1.7)^5 - *growth rate doubling 5 times*
t(n) = 74.5 for 1M ints

These estimates are low, probably because these estimations predict much smaller growth rate than the O(n^2) that TwoSum has.

2.) TwoSumFast.java estimates:

4K to 8K
.007/.003 ~ 2 times larger
8K to 16K
.02/.007 ~ 3 times larger

16K to 32K
t(n) / .02 = 1.5(3) times larger *(estimate based on growth rate of 3/2 = 1.5)*

t(n) = .09 for 32K ints

32K to 1M
31.25~(2^5) (*doubles about five times)*
t(n)/.09 = 4.5 * (1.5)^5 - *growth rate doubling 5 times*
t(n) = 3.1 for 1M ints

These estimates are a little high, maybe because these estimations are predicted based on doubling sizes, not the growth rate of O(n log(n)).

3.)ThreeSum.java estimates:
1K to 2K ~ 7 times larger
2K to 4K ~10 times larger
4K to 8K ~ 10 times larger
8K to 16K~ 7.5 times larger

16K to 32K
t(n) / 702.9 = 8 times larger *(estimate based on growth rate of 3/2 = 1.5)*
t(n) = 5623.2 for 32K ints

32K to 1M
31.25~(2^5) (*doubles about five times)*
t(n)/5623.2 = (8)^5 - *growth rate doubling 5 times*
t(n) = 184261020 for 1M ints

These estimates might be accurate, because the factor of growth seems relatively constant, no actual runtimes could be obtained for these values.

4.)ThreeSumFast.java estimates:
2K to 4K ~ 3 times
4K to 8K ~ 4 times larger
8K to 16K~4 times larger

16K to 32K
t(n) / 8.2 = 4 times larger *(estimate based on growth rate of 3/2 = 1.5)*
t(n) = 32.8 for 32K ints

32K to 1M
31.25~(2^5) (*doubles about five times)*
t(n)/32.8 = (4)^5 - *growth rate doubling 5 times*
t(n) = 33587 for 1M ints

These estimates seem plausible as well, because the growth rate is fairly constant. The 32K ints estimate is quite accurate, and no value was able to be obtained for 1M ints.