Lab 2 Report - Analysis of Algorithms

Dane Backbier

Task 1

1) Order of Growth for the Running Time of Count Function in TwoSum.java

```
    public static int count(int[] a) {
        int n = a.length; // O(1), assignment
        int count = 0; // O(1), assignment
        for (int i = 0; i < n; i++) { // loops n times, O(n), and O(1) assignment operation inside
            for (int j = i+1; j < n; j++) { // loops n times, O(n), and O(1) assignment operation inside
                if (a[i] + a[j] == 0) { // O(1)
                    count++; // O(1) (only executed when condition is true)
                }
            }
        }
        return count; // O(1)
    }  // O(n) * O(n) = O(n^2), constant time operations can be ignored for Big-Oh notation
       // n^2 from two nested loops each going up to n
```

2) Order of Growth for the Running Time of Count Function in TwoSumFast.java

```
    public static int count(int[] a) {
        int n = a.length; // O(1), assignment
        Arrays.sort(a); // O(n log n)
        if (containsDuplicates(a)) throw new IllegalArgumentException("array contains duplicate integers"); // O(n) from containsDuplicates call
        int count = 0; // O(1), assignment
        for (int i = 0; i < n; i++) { // loops n times, O(n), and O(1) assignment operation inside
            int j = Arrays.binarySearch(a, -a[i]); // O(log n)
            if (j > i) count++; // O(1) (only executed when condition is true)
        }
```

```
        return count; // O(1)
    } // O(n) * O(log n) = O(n log n), constant time operations can be
ignored for Big-Oh notation
```

3) Order of Growth for the Running Time of Count Function in ThreeSum.java

```
    public static int count(int[] a) {
        int n = a.length; // O(1), assignment
        int count = 0; // O(1), assignment
        for (int i = 0; i < n; i++) { // loops n times, O(n), and O(1)
assignment operation inside
            for (int j = i+1; j < n; j++) { // loops n times, O(n), and
O(1) assignment operation inside
                for (int k = j+1; k < n; k++) { // loops n times, O(n),
and O(1) assignment operation inside
                    if (a[i] + a[j] + a[k] == 0) { // O(1)
                        count++; // O(1), only executed when condition is
true
                    }
                }
            }
        }
        return count; // O(1)
    } // O(n) * O(n) * O(n) = O(n^3), constant time operations can be
ignored for Big-Oh notation
      // n^3 from three nested loops each going up to n
```

4) Order of Growth for the Running Time of Count Function in ThreeSumFast.java

```
    public static int count(int[] a) {
        int n = a.length; // O(1), assignment
        Arrays.sort(a); // O(n log n), from the sort() function call
        if (containsDuplicates(a)) throw new
IllegalArgumentException("array contains duplicate integers"); // O(n),
from containsDuplicates call
        int count = 0; // O(1), assignment
        for (int i = 0; i < n; i++) { // loops n times, O(n), and O(1)
assignment operation inside
```

```
            for (int j = i+1; j < n; j++) { // loops n times, O(n), and
O(1) assignment operation inside
                int k = Arrays.binarySearch(a, -(a[i] + a[j])); // O(log
n)
                if (k > j) count++; // O(1) (only executed when condition
is true)
            }
        }
        return count; // O(1)
    } // O(n) * O(n) * O(log n) = O(n^2 log n), constant time operations
can be ignored for Big-Oh notation
```

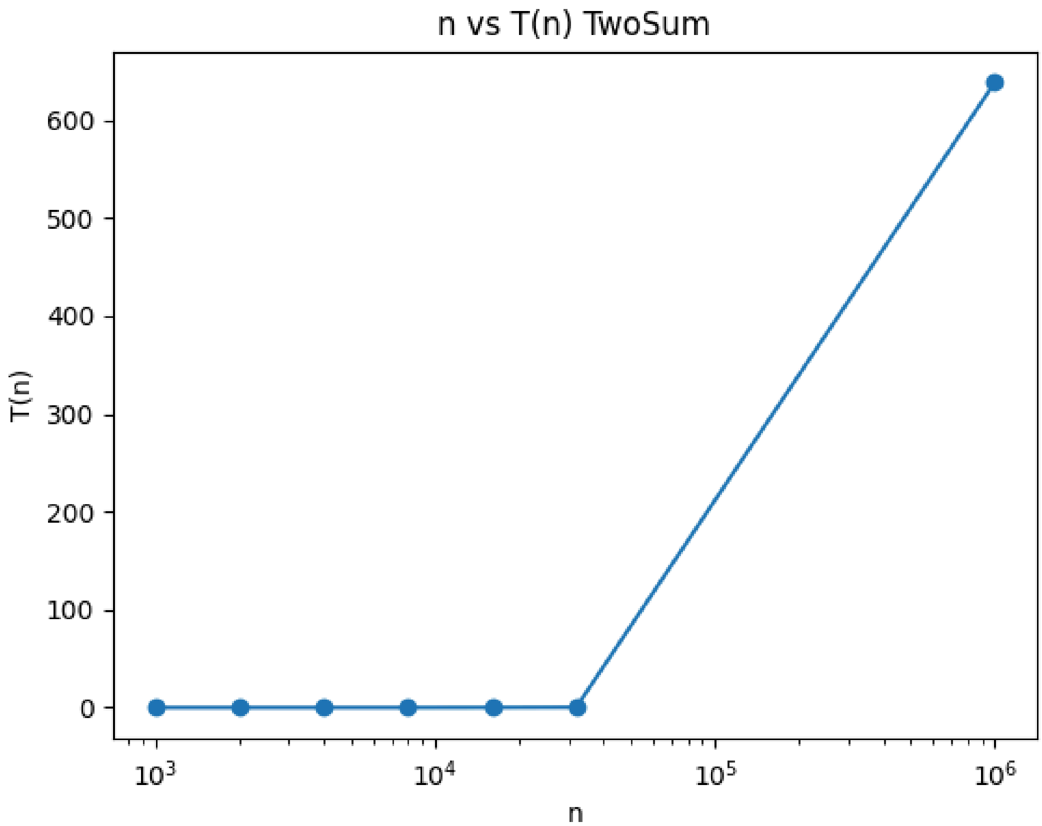
Task 2

1) TwoSum.java
    a) Output: (separate because screenshots were taken at different times)

```
C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSum 1Kints.txt
      1     0.0   20250908_143422  dbackbie  1Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSum 2Kints.txt
      2     0.0   20250908_143535  dbackbie  2Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSum 4Kints.txt
      3     0.0   20250908_143547  dbackbie  4Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSum 8Kints.txt
     19     0.0   20250908_143556  dbackbie  8Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSum 16Kints.txt
     66     0.1   20250908_143603  dbackbie  16Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSum 32Kints.txt
    273     0.3   20250908_143610  dbackbie  32Kints.txt
```

```
c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSum 1Mints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
 249838   638.5   20250909_120950  dbackbie  1Mints.txt
```

b) Graph #1: n vs. T(n) TwoSum

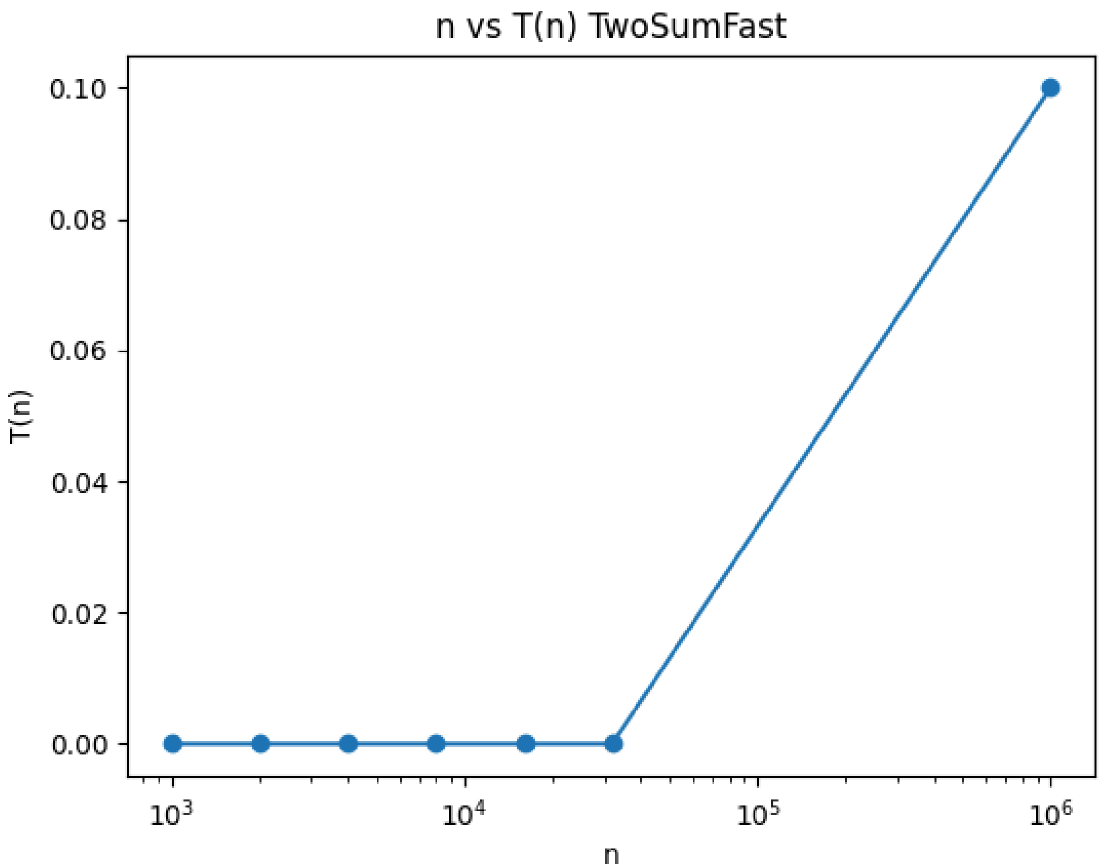


2) TwoSumFast.java

a) Output:

```
C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSumFast 1Kints.txt
      1     0.0   20250908_145754  dbackbie  1Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSumFast 2Kints.txt
      2     0.0   20250908_145814  dbackbie  2Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSumFast 4Kints.txt
      3     0.0   20250908_145827  dbackbie  4Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSumFast 8Kints.txt
     19     0.0   20250908_145836  dbackbie  8Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSumFast 16Kints.txt
     66     0.0   20250908_145846  dbackbie  16Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSumFast 32Kints.txt
    273     0.0   20250908_145933  dbackbie  32Kints.txt

C:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java TwoSumFast 1Mints.txt
 249838     0.1   20250908_145947  dbackbie  1Mints.txt
```

b) Graph #2: n vs. T(n) TwoSumFast



3) ThreeSum.java

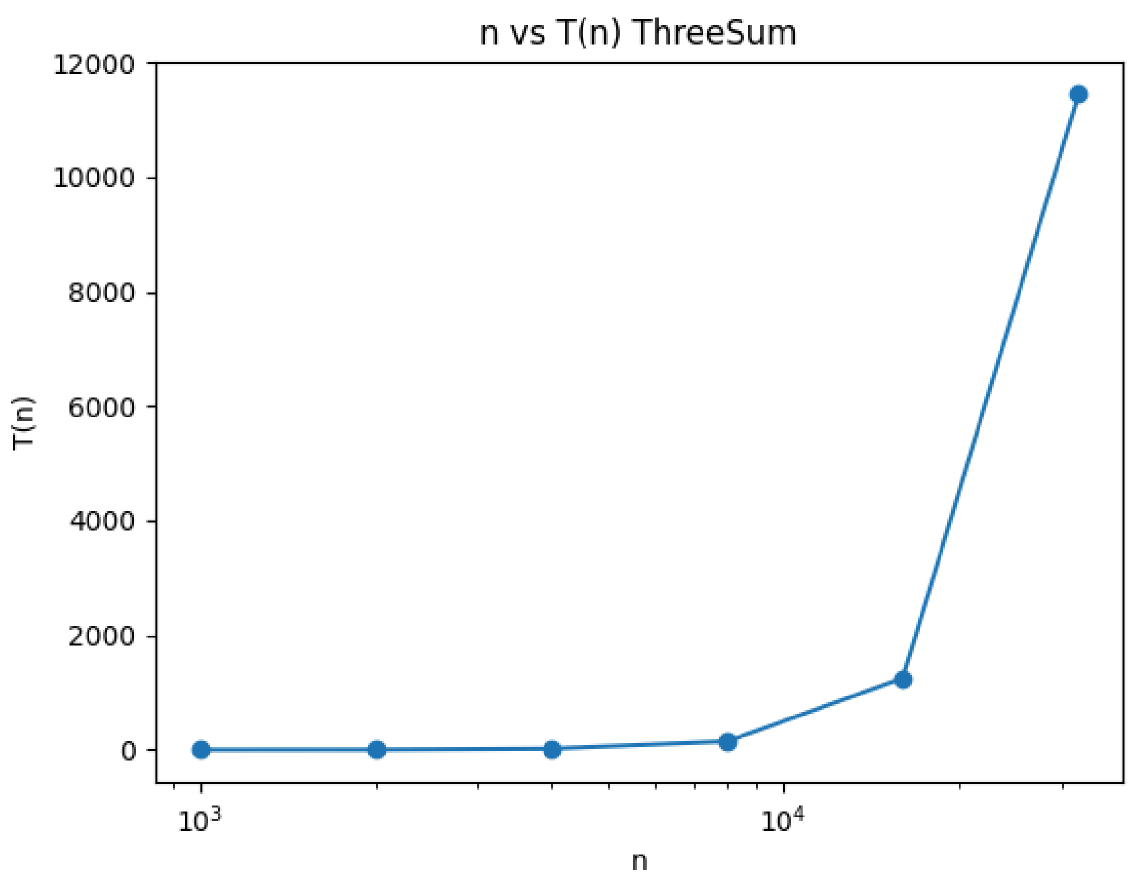
a) Output:

```
c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSum 1Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
     70     0.3   20250909_171731  dbackbie  1Kints.txt

c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSum 2Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
    528     2.3   20250909_171741  dbackbie  2Kints.txt

c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSum 4Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
   4039    19.0   20250909_171807  dbackbie  4Kints.txt

c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSum 8Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
  32074   146.1   20250909_172039  dbackbie  8Kints.txt
```

```
c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSum 16Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
 255181  1247.0   20250909_174554  dbackbie  16Kints.txt
```

```
c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSum 32Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
2052358 11443.1   20250909_210733  dbackbie  32Kints.txt
```

b) Graph #3: n vs. T(n) ThreeSum

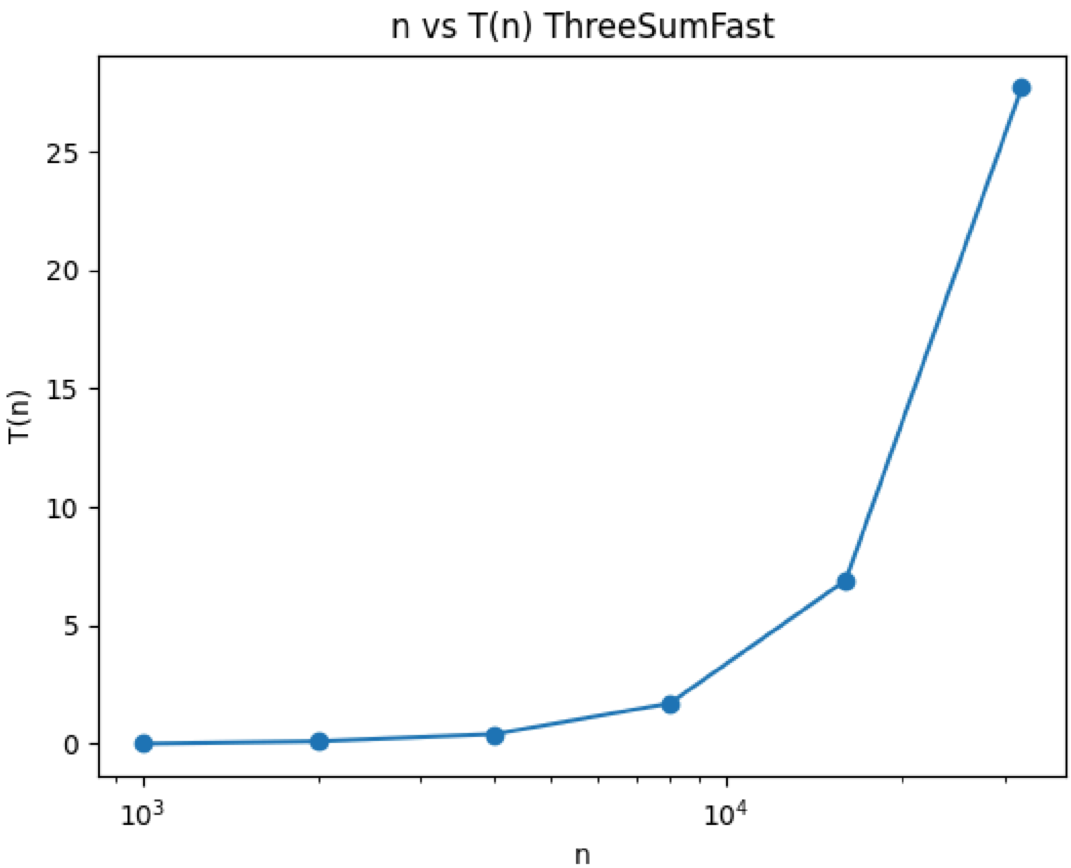


4) ThreeSumFast.java

a) Output:

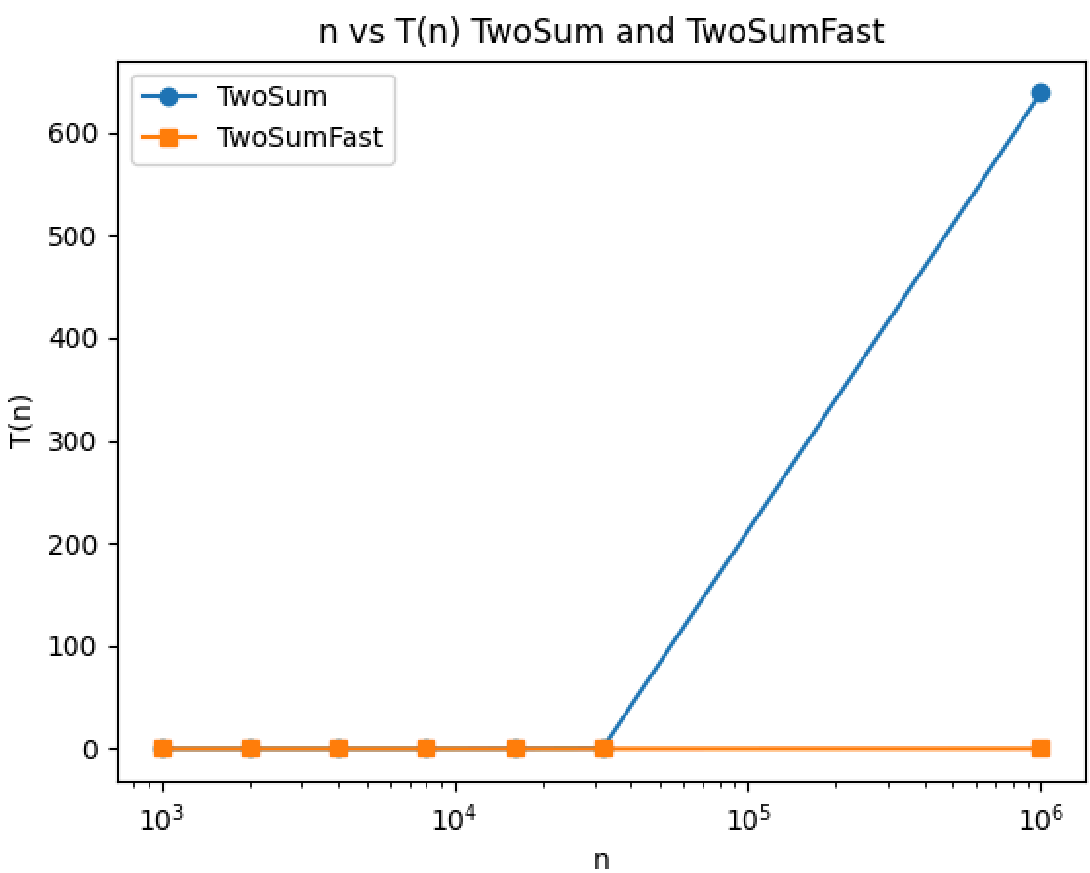
```
c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSumFast 1Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
     70     0.0   20250909_211438  dbackbie  1Kints.txt

c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSumFast 2Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
    528     0.1   20250909_211444  dbackbie  2Kints.txt

c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSumFast 4Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
   4039     0.4   20250909_211451  dbackbie  4Kints.txt

c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSumFast 8Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
  32074     1.7   20250909_211459  dbackbie  8Kints.txt

c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSumFast 16Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
 255181     6.9   20250909_211512  dbackbie  16Kints.txt

c:\Users\daneb\.vscode\CSC 172\Labs\Lab2_DB>java ThreeSumFast 32Kints.txt
Picked up JAVA_TOOL_OPTIONS: -Dstdout.encoding=UTF-8 -Dstderr.encoding=UTF-8
2052358    27.7   20250909_211546  dbackbie  32Kints.txt
```

b) Graph #4: n vs. T(n) ThreeSumFast



5) TwoSum.java & TwoSumFast.java

a) Graph #5: n vs. T(n) TwoSum and TwoSumFast

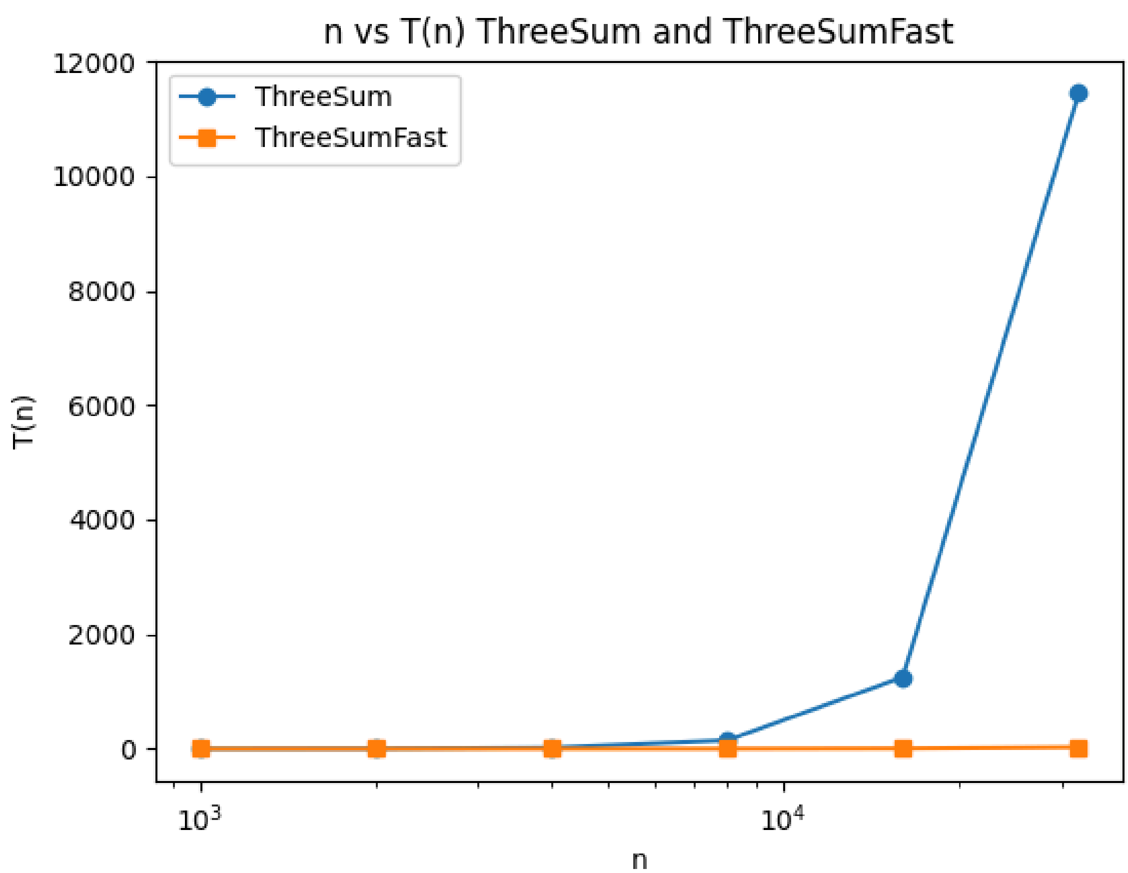


b) Description of Findings:

Graph #5 compares the time it took for the TwoSum and TwoSumFast algorithms to run on different-sized inputs. It shows how much more effective TwoSumFast is for inputs larger than 16,000 integers compared to TwoSum. However, for smaller inputs, such as 1,000 to 8,000 integers, the differences in runtime are negligible. In practice, however, inputs can commonly reach lengths in the millions or billions. Therefore, the algorithm that would actually be implemented would be TwoSumFast for its capabilities to handle large inputs.

6) ThreeSum.java & ThreeSumFast.java

a) Graph #6: n vs. T(n) ThreeSum and ThreeSumFast



b) Description of Findings:

Graph #6 compares the runtime for the ThreeSum and ThreeSumFast algorithms for different-sized inputs. For the first 3 inputs, 1,000, 2,000, and 4,000 integers, there's a negligible difference in T(n) between the algorithms. However, for the rest of the inputs, the runtime for ThreeSum begins to increase exponentially, while the runtime for ThreeSumFast, although increasing, it increases so slightly that it can't be seen through the graph.

Task 3

1) Comparing Runtimes for TwoSum.java

Table #1: TwoSum.java

	TwoSum.java
n	T(n) (seconds)
1000	0
2000	0
4000	0
8000	0

16000	0.1
32000	0.3
1000000	638.5

The runtimes for n = 1000 and n = 2000 are certainly not both zero; however, they are both so short that the computer's rounding makes them appear to be zero. Same with n = 4000 and n = 8000. However, from n = 8000 to n = 16000, the runtime increases from 0.0 to 0.1 seconds. This may allow us to estimate the runtime for n = 32000 by noticing that the input size doubles from 16000 to 32000. Since the time complexity of TwoSum.java is O(n^2), a two times increase in input size would result in a 4 times increase in runtime ($2^2 = 4$), resulting in a runtime of 0.4. This is pretty close to 0.3, especially when we don't see the rest of the decimal points, as it could be 0.37 or 0.38. This overestimation could be because Big-Oh notation assumes the worst possible scenario. We are also able to estimate the runtime for when n = 1000000. This can be done by finding the quotient of $\frac{1000000}{32000} = 31.25$. Using the same strategy as before, $31.25^2 = 976.6$, and $976.6 \times 0.3 = 293.0$. This is nowhere close to the actual runtime, but this could be due to the computer having background processes taking up priority.

2) Comparing Runtimes for TwoSumFast.java

Table #2: TwoSumFast.java

	TwoSumFast.java
n	T(n) (seconds)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	0
1000000	0.1

Similarly to TwoSum.java, the runtime for TwoSumFast.java is so short that the computer's rounding makes them appear to be zero. But, this is the situation for all inputs from n = 1000 to n = 16000. This makes it impossible to estimate the runtimes for when n = 32000 or when n = 1000000. However, this indicates how effective TwoSumFast.java is at computing very large inputs, especially compared to TwoSum.java.

3) Comparing Runtimes for ThreeSum.java

Table #3: ThreeSum.java (did not use 1000000 because it would take too long)

	ThreeSum.java
n	T(n) (seconds)

1000	0.3
2000	2.3
4000	19
8000	146
16000	1247
32000	11443.1

The runtimes for ThreeSum.java are much easier to compare than TwoSum.java or TwoSumFast.java, as it takes long enough for the computer's rounding not to make it zero. Hence, when n increases from 1000 to 2000, the runtimes increase by ~2 seconds, and when n increases by 4000 (from 4000 to 8000), runtimes increase from 19 seconds to 146 seconds. Then, when n increases from 8000 to 16000, you could begin to notice how the time complexity is cubic (O(n^3)). This is because the factor by which the runtimes are increasing is always ~8. This makes it cubic because the inputs are doubling with every step, and 2^3 = 8. Using the data in Table #3, it's possible to estimate the runtimes for when n = 32000 and n = 1000000. Since ThreeSum.java is O(n^3), you just need to multiply the previous runtime by the factor by which the inputs increase to estimate the runtime. So, $1247 \times 8 = 9976$ seconds, and using the previous estimation to estimate when n = 1000000, $9976 \times (\frac{1000000}{32000})^3 = 304443359$ seconds. That is equivalent to close to 10 years' worth of runtime, and using the actual runtime to estimate when n = 1000000, it would take a little longer.

4) Comparing Runtimes for ThreeSumFast.java

Table #4: ThreeSumFast.java (did not use 1000000 because ThreeSum.java didn't use it)

	ThreeSumFast.java
n	T(n) (seconds)
1000	0
2000	0.1
4000	0.4
8000	1.7
16000	6.9
32000	27.7

Finally, for ThreeSumFast.java, the time complexity is O(n^2 log n). Comparing n = 1000 and n = 2000, the runtimes increase from 0.0 to 0.1 seconds, then from n = 4000 to n = 8000, the runtimes increase from 0.4 to 1.7 seconds, and from n = 8000 to n = 16000, the runtimes increase from 1.7 to 6.9 seconds. The factor by which the runtimes increase is therefore around 4 ($\frac{1.7}{0.4} = 4.25$ and $\frac{6.9}{1.7} = 4.06$), this makes sense because inputting 2 (factor by which inputs are increasing) into O(n^2 log n) = $2^2 \, log_2(2) = 4$. Therefore, we can estimate for when n = 32000 in $4 \times 6.9 = 27.6$ seconds. This is very close to the real value of 27.7. Additionally, it's also possible to estimate for when n = 1000000, by $31.25^2 \log(31.25) = 4849$. Now to estimate for when n = 1000000 through $4849 \times 27.6 = 133841$. This is equivalent to a little over 1.5 days.

Appendix A - Program Creating Graphs 1-4: plotOneLine.py

```
import matplotlib.pyplot as plt

x = [1000, 2000, 4000, 8000, 16000, 32000, 1000000] # 1Mints.txt won't be
used for ThreeSum and ThreeSumFast because it will take too long
y = [0.0, 0.0, 0.0, 0.0, 0.1, 0.3, 638.5] # Data from TwoSum algorithm

plt.plot(x, y, marker="o", linestyle="-")

plt.xscale("log") # scale n values logarithmically

plt.xlabel("n")
plt.ylabel("T(n)")
plt.title("n vs T(n) TwoSum") #Title for TwoSum, changes depending on
which algorithm is being graphed

plt.show()
```

Appendix B - Program Creating Graphs 5 & 6: plotTwoLine.py

```
import matplotlib.pyplot as plt

x = [1000, 2000, 4000, 8000, 16000, 32000, 1000000] # 1Mints.txt won't be
used for ThreeSum and ThreeSumFast because it will take too long
y1 = [0.0, 0.0, 0.0, 0.0, 0.1, 0.3, 638.5] # Data from TwoSum algorithm
y2 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.1] # Data from TwoSumFast algorithm

plt.plot(x, y1, marker="o", label="y = x²")    # first line, using y1
values
plt.plot(x, y2, marker="s", label="y = x")     # second line, using y2
values

plt.xscale("log") # scale n values logarithmically

plt.xlabel("n")
plt.ylabel("T(n)")
```

```
plt.title("n vs T(n) TwoSum and TwoSumFast") # Title for TwoSum and
TwoSumFast, changes depending on which algorithms are being graphed

plt.show()
```