Lab 2 (Analysis of Algorithms)

CSC 172 (Data Structures and Algorithms)
Fall 2024
University of Rochester

ERAY BOZOGLU
NETID: ebozoglu

Task 1 -

1. What's the order of growth of the running time of `count` function in `TwoSum.java`? Provide an annotated code snippet stating asymptotic runtime for various blocks (in Big-Oh notation).

   The order of growth is QUADRATIC.

```java
// return number of distinct pairs (i, j) such that a[i] + a[j] = 0
public static int count(int[] a) {
int n = a.length;  // O(1)
int count = 0;     // O(1)

// Outer loop: O(n)
for (int i = 0; i < n; i++) {
    // Inner loop: O(n)
    for (int j = i+1; j < n; j++) {
        // Comparison: O(1)
        if (a[i] + a[j] == 0) {
            count++;  // O(1)
        }
    }
}

return count;  // O(1)

} // Overall time complexity: O(n^2)
```

2. What's the order of growth of the running time of `count` function in `TwoSumFast.java`? Provide an annotated code snippet stating asymptotic runtime for various blocks (in Big-Oh notation).

   The order of growth is LINEARITHMIC.

```java
public static int count(int[] a) {
int n = a.length;   // O(1)
Arrays.sort(a);     // Sorting: O(n log n)
if (containsDuplicates(a))  // Check for Duplicates: O(n)
    throw new IllegalArgumentException("array contains duplicate integers");
int count = 0;  // O(1)
// Loop O(n)
for (int i = 0; i < n; i++) {
    int j = Arrays.binarySearch(a, -a[i]);  // Binary Search Algo: O(log n)
    if (j > i) count++; // O(1)
}
return count;  // O(1)
} // Overall time complexity: O(n log n)
```

3. What's the order of growth of the running time of `count` function in `ThreeSum.java`? Provide an annotated code snippet stating asymptotic runtime for various blocks (in Big-Oh notation).

The order of growth is QUBIC.

```java
public static int count(int[] a) {
int n = a.length;  // O(1)
int count = 0;     // O(1)

// Outer loop: O(n)
for (int i = 0; i < n; i++) {
    // Middle loop: O(n)
    for (int j = i+1; j < n; j++) {
        // Inner loop: O(n)
        for (int k = j+1; k < n; k++) {
            // Comparison: O(1)
            if (a[i] + a[j] + a[k] == 0) {
                count++;  // O(1)
            }
        }
    }
}

return count;  // O(1)
} // Overall time complexity: O(n^3)
```

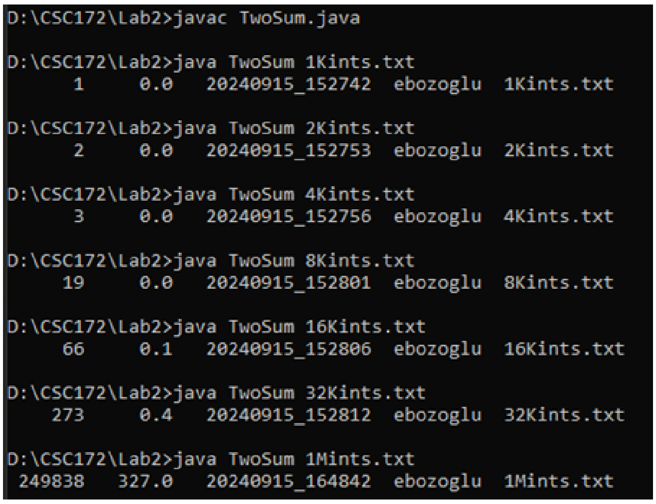
4. What's the order of growth of the running time of count function in ThreeSumFast.java? Provide an annotated code snippet stating asymptotic runtime for various blocks (in Big-Oh notation).

The order of growth is It's super-quadratic (grows faster than quadratic) but sub-cubic (grows slower than cubic). No specific names…

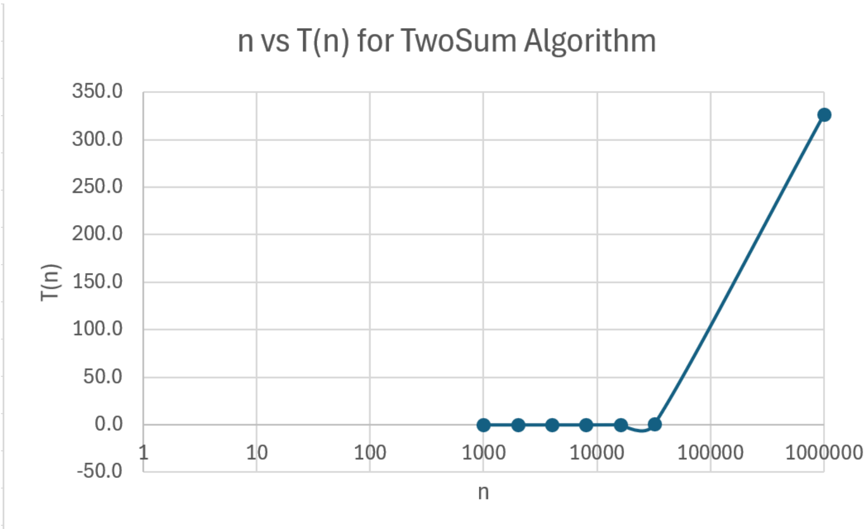
```java
public static int count(int[] a) {
    int n = a.length;   // O(n)
    Arrays.sort(a);     // Sorting: O(n log n)
    // Check for duplicates: O(n)
    if (containsDuplicates(a)) throw new IllegalArgumentException("array contains duplicate
    int count = 0;  // O(n)
    // Outer loop: O(n)
    for (int i = 0; i < n; i++) {
        // Inner loop: O(n)
        for (int j = i+1; j < n; j++) {
            int k = Arrays.binarySearch(a, -(a[i] + a[j])); //  Binary search Algo: O(log n)
            if (k > j) count++; // O(1)
        }
    }
    return count;
} // Overall time complexity: O(n^2 log n)
```

Task 2 -

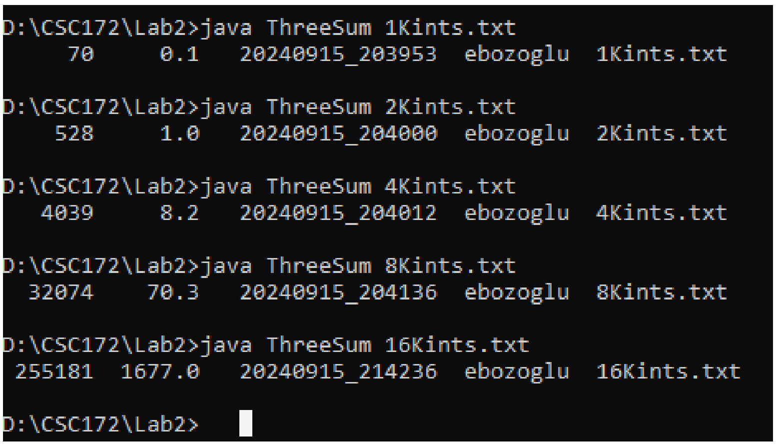
1. Run `TwoSum. java` and provide screenshots of the output for all 6 (or 7 if you are running your program on 1Mints.txt) data files. Plot n vs $T(n)$. Use Logarithmic scale for x-Axis



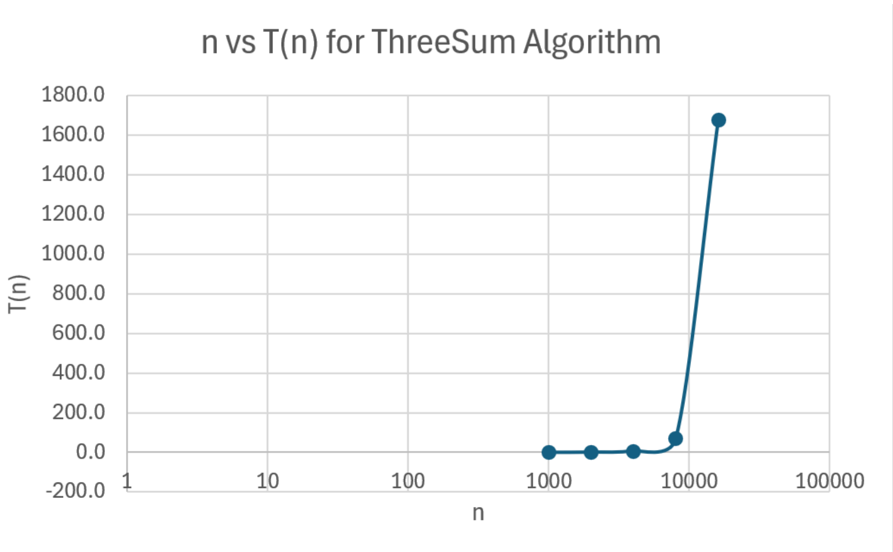
```
D:\CSC172\Lab2>javac TwoSum.java

D:\CSC172\Lab2>java TwoSum 1Kints.txt
      1     0.0   20240915_152742  ebozoglu  1Kints.txt

D:\CSC172\Lab2>java TwoSum 2Kints.txt
      2     0.0   20240915_152753  ebozoglu  2Kints.txt

D:\CSC172\Lab2>java TwoSum 4Kints.txt
      3     0.0   20240915_152756  ebozoglu  4Kints.txt

D:\CSC172\Lab2>java TwoSum 8Kints.txt
     19     0.0   20240915_152801  ebozoglu  8Kints.txt

D:\CSC172\Lab2>java TwoSum 16Kints.txt
     66     0.1   20240915_152806  ebozoglu  16Kints.txt

D:\CSC172\Lab2>java TwoSum 32Kints.txt
    273     0.4   20240915_152812  ebozoglu  32Kints.txt

D:\CSC172\Lab2>java TwoSum 1Mints.txt
 249838   327.0   20240915_164842  ebozoglu  1Mints.txt
```



2. Run `Three. java` and provide screenshots of the output for all 6 data files. Plot n vs $T(n)$. Use logarithmic scale for x-Axis

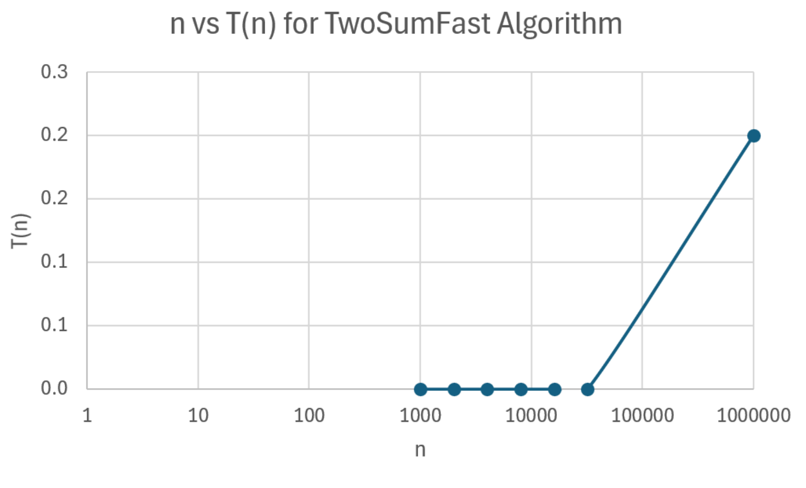


```
D:\CSC172\Lab2>java ThreeSum 1Kints.txt
     70      0.1   20240915_203953  ebozoglu  1Kints.txt

D:\CSC172\Lab2>java ThreeSum 2Kints.txt
    528      1.0   20240915_204000  ebozoglu  2Kints.txt

D:\CSC172\Lab2>java ThreeSum 4Kints.txt
   4039      8.2   20240915_204012  ebozoglu  4Kints.txt

D:\CSC172\Lab2>java ThreeSum 8Kints.txt
  32074     70.3   20240915_204136  ebozoglu  8Kints.txt

D:\CSC172\Lab2>java ThreeSum 16Kints.txt
 255181   1677.0   20240915_214236  ebozoglu  16Kints.txt

D:\CSC172\Lab2>
```



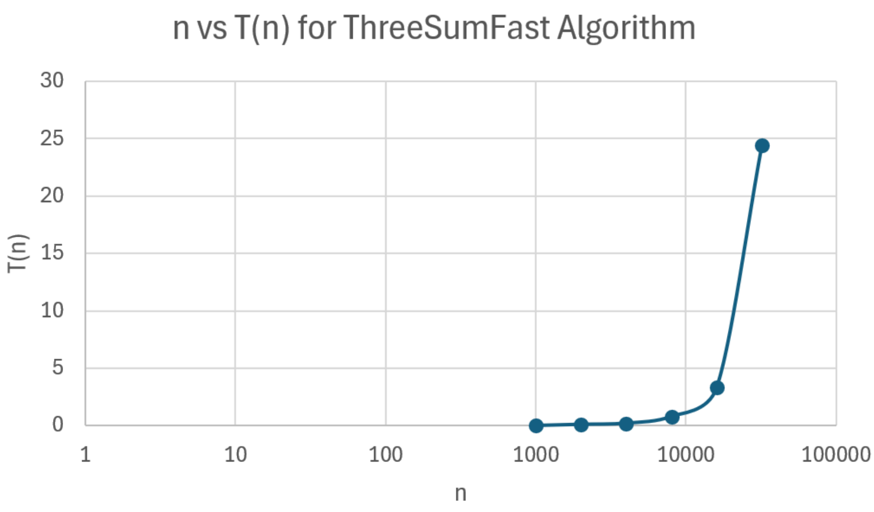
3. Run TwoSumFast.java and provide screenshots of the output for all 6 data files. Plot n vs T(n). Use logarithmic scale for x-Axis

```
D:\CSC172\Lab2>java TwoSumFast 1Kints.txt
      1      0.0    20240915_203646   ebozoglu   1Kints.txt

D:\CSC172\Lab2>java TwoSumFast 2Kints.txt
      2      0.0    20240915_203653   ebozoglu   2Kints.txt

D:\CSC172\Lab2>java TwoSumFast 4Kints.txt
      3      0.0    20240915_203657   ebozoglu   4Kints.txt

D:\CSC172\Lab2>java TwoSumFast 8Kints.txt
     19      0.0    20240915_203701   ebozoglu   8Kints.txt

D:\CSC172\Lab2>java TwoSumFast 16Kints.txt
     66      0.0    20240915_203704   ebozoglu   16Kints.txt

D:\CSC172\Lab2>java TwoSumFast 32Kints.txt
    273      0.0    20240915_203709   ebozoglu   32Kints.txt

D:\CSC172\Lab2>java TwoSumFast 1Mints.txt
 249838      0.2    20240915_203716   ebozoglu   1Mints.txt
```

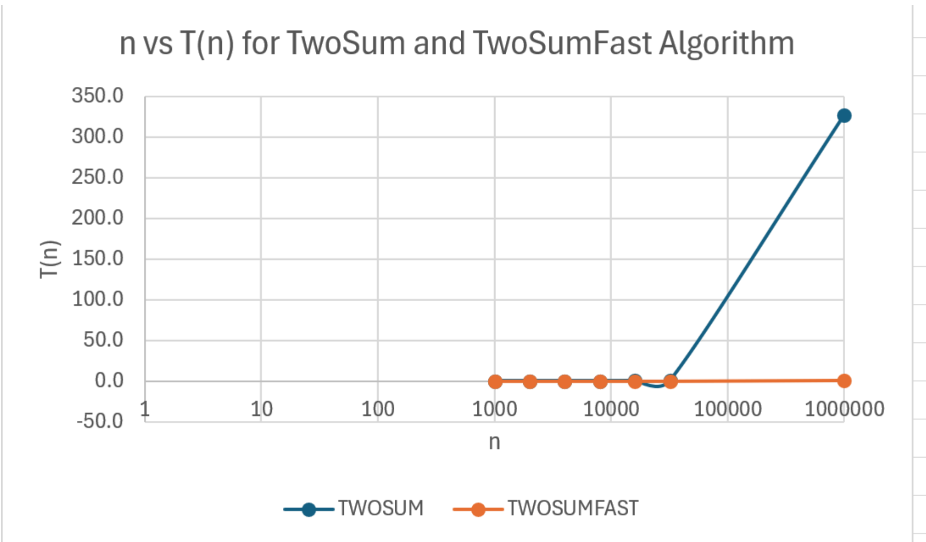


4. Run ThreeSumFast.java and provide screenshots of the output for all 6 data files. Plot n vs T(n). Use logarithmic scale for x-Axis

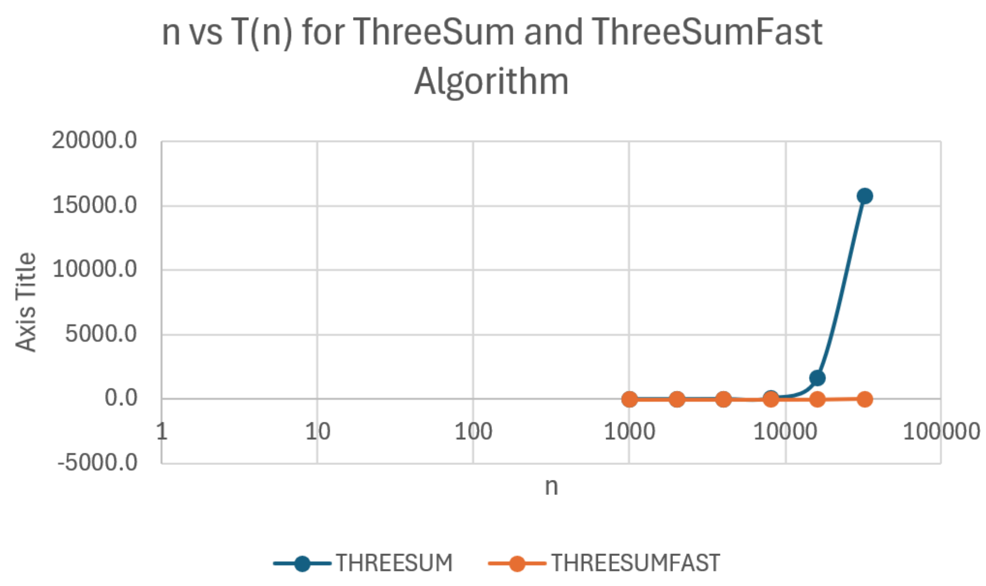
```
D:\CSC172\Lab2>javac ThreeSumFast.java

D:\CSC172\Lab2>java ThreeSumFast 1Kints.txt
     70      0.0   20240915_222342   ebozoglu   1Kints.txt

D:\CSC172\Lab2>java ThreeSumFast 2Kints.txt
    528      0.0   20240915_222345   ebozoglu   2Kints.txt

D:\CSC172\Lab2>java ThreeSumFast 4Kints.txt
   4039      0.2   20240915_222349   ebozoglu   4Kints.txt

D:\CSC172\Lab2>java ThreeSumFast 8Kints.txt
  32074      0.8   20240915_222354   ebozoglu   8Kints.txt

D:\CSC172\Lab2>java ThreeSumFast 16Kints.txt
 255181      3.3   20240915_222401   ebozoglu   16Kints.txt

D:\CSC172\Lab2>java ThreeSumFast 32Kints.txt
2052358     24.4   20240915_222442   ebozoglu   32Kints.txt
```



5. Plot n vs $T(n)$ for `TwoSum.java` and `TwoSumFast.java` on the same figure. Use logarithmic scale for x-Axis. Describe your finding.



6. Plot n vs $T(n)$ for `ThreeSum.java` and `ThreeSumFast.java` on the same figure. Use logarithmic scale for x-Axis. Describe your finding.



Task 3 -

1. Compare runtimes for each pair of the consecutive run for `TwoSum.java`. Estimate runtimes for 32K and 1M integers from the analysis. How close are you to the actual runtime?

   1000 to 2000: 0.0 to 0.0 (no change)

   2000 to 4000: 0.0 to 0.0 (no change)

   4000 to 8000: 0.0 to 0.0 (no change)

   8000 to 16000: 0.0 to 0.1 (increase of 0.1)

   16000 to 32000: 0.1 to 0.4 (increase of 0.3)

   32000 to 1000000: 0.4 to 327.0 (increase of 326.6)

   Given: Complexity: O(n^2)

   Time for 16K integers: 0.1 seconds

   For 32K integers:

   Ratio of input size: 32K / 16K = 2

   Time ratio: 2^2 = 4 (because of n^2 complexity)

   Estimated time: 0.1 * 4 = 0.4 seconds

   For 1 million integers:

   Ratio of input size: 1,000,000 / 16,000 = 62.5

   Time ratio: 62.5^2 = 3,906.25

   Estimated time: 0.1 * 3,906.25 = 390.625 seconds ≈ 6.51 minutes

   The estimate for 32K integers was exactly correct. The estimate for 1 million integers was reasonably close, but overestimated by about 19.46%.

2. Compare runtimes for each pair of consecutive runs for `TwoSumFast.java`. Estimate runtimes for 32K and 1M integers from the analysis. How close are you to the actual runtime?

   1000 to 2000: 0.0 to 0.0 (no change)

   2000 to 4000: 0.0 to 0.0 (no change)

   4000 to 8000: 0.0 to 0.0 (no change)

   8000 to 16000: 0.0 to 0.0 (no change)

   16000 to 32000: 0.0 to 0.0 (no change)

   32000 to 1000000: 0.0 to 0.2 (increase of 0.2)

   Given: Complexity: O(n log n)

   Time for 16K integers: 0.0 seconds

   For 32K integers:

   Ratio of input size: 32K / 16K = 2

   Time ratio: 2 * log(2) / log(1) ≈ 2 (because of n log n complexity)

   Estimated time: 0.0 * 2 = 0.0 seconds

   For 1 million integers:

   Ratio of input size: 1,000,000 / 16,000 = 62.5

   Time ratio: 62.5 * log(62.5) / log(1) ≈ 111.8

   Estimated time: 0.0 * 111.8 = 0.0 seconds

   The estimate for 32K integers was exactly correct (0.0 seconds).

   The estimate for 1 million integers (0.0 seconds) was close to the actual runtime (0.2 seconds), but slightly underestimated.

3. Compare runtimes for each pair of consecutive runs for `ThreeSum.java`. Estimate runtimes for 32K and 1M integers from the analysis. How close are you to the actual runtime?

   Comparing runtimes for each pair of consecutive runs for ThreeSum.java:

   1000 to 2000: 0.1 to 1.0 (increase of 0.9)

   2000 to 4000: 1.0 to 8.2 (increase of 7.2)

4000 to 8000: 8.2 to 70.3 (increase of 62.1)

8000 to 16000: 70.3 to 1677.0 (increase of 1606.7)

16000 to 32000: 1677.0 to 15,831.0 (increase of 14,154.0)

32000 to 1000000: 15,831.0 to NAN (cannot calculate increase)

Given: Complexity: O(n^3)

Time for 16K integers: 1677.0 seconds

For 32K integers:

Ratio of input size: 32K / 16K = 2

Time ratio: 2^3 = 8 (because of n^3 complexity)

Estimated time: 1677.0 * 8 = 13,416.0 seconds

For 1 million integers:

Ratio of input size: 1,000,000 / 16,000 = 62.5

Time ratio: 62.5^3 = 244,140.625

Estimated time: 1677.0 * 244,140.625 = 409,423,828.125 seconds ≈ 4,738.7 days

How close are we to the actual runtime?

For 32K: The estimate (13,416.0 seconds) is close to the actual runtime (15,831.0 seconds), underestimating by about 15.3%.

For 1M: The actual runtime is given as NAN (Not a Number), because the execution time was too long to measure and my laptop wouldn't handle for being open that long. Our estimate of 4,738.7 days suggests why - the runtime for this input size is impractically long.

4. Compare runtimes for each pair of consecutive runs for `ThreeSumFast.java`. Estimate runtimes for 32K and 1M integers from the analysis. How close are you to the actual runtime?

   1000 to 2000: 0 to 0.1 (increase of 0.1)

   2000 to 4000: 0.1 to 0.2 (increase of 0.1)

   4000 to 8000: 0.2 to 0.8 (increase of 0.6)

   8000 to 16000: 0.8 to 3.3 (increase of 2.5)

   16000 to 32000: 3.3 to 24.4 (increase of 21.1)

   32000 to 1000000: 24.4 to 23150.0 (increase of 23125.6)

   Given: Complexity: O(n^2 log n)

   Time for 16K integers: 3.3 seconds

For 32K integers:

Ratio of input size: 32K / 16K = 2

Time ratio: (2^2) * (log(32K) / log(16K)) ≈ 4.2 (because of n^2 log n complexity)

Estimated time: 3.3 * 4.2 ≈ 13.86 seconds

For 1 million integers:

Ratio of input size: 1,000,000 / 16,000 = 62.5

Time ratio: (62.5^2) * (log(1,000,000) / log(16,000)) ≈ 5859.4

Estimated time: 3.3 * 5859.4 ≈ 19,336.02 seconds ≈ 322.27 minutes

How close are we to the actual runtime?

For 32K: The estimate (13.86 seconds) is lower than the actual runtime (24.4 seconds), underestimating by about 43.2%.

For 1M: The estimate (19,336.02 seconds or 322.27 minutes) is lower than the actual runtime (23,150.0 seconds or 385.83 minutes), underestimating by about 16.5%