Project 1 - Asymptotic Analysis

Author: Siddhesh Parab

1 Problem Statement

We are required to analyze the following program/code(Option 2) sample. int j

```
= 2

while (j < n) {

  k = 2

  while (k < n) { Sum

    += a[k]*b[k] k = k

    * sqrt(k)

  }

  j = j + j/2

}
```

2 Theoretical Analysis

To determine the overall time complexity, we need to analyze both the inner and outer loops independently and then combine their complexities.
Inner Loop Analysis

The inner loop's control variable is k, which is initialized to 2 and updated by k = k * sqrt(k). This can be rewritten as k = k^(3/2). Let's trace the values of k:

- Iteration 1: k = 2
- Iteration 2: k = 2^(3/2)
- Iteration 3: k = (2^(3/2))^(3/2) = 2^((3/2)^2)
- Iteration m: k = 2^((3/2)^(m-1))

The inner loop terminates when k >= n. So, we need to find m such that:
2^((3/2)^(m-1)) >= nTaking log (base 2) on both sides:

$(3/2)^{(m-1)} >= \log_2(n)$Taking log (base 3/2) on both sides:

m-1 $>= \log_{3/2}(\log_2(n))$m $>= \log_{3/2}(\log_2(n)) + 1$Therefore, the inner loop runs approximately O(log(log n)) times.

Iteration Values Table

Iteration	Outer Loop (j)	Inner Loop (k)
1	2	2
2	3	2^(3/2)
3	4.5	2^((3/2)^2)
...	...	...
p	2 * (3/2)^(p-1)	
m		2^((3/2)^(m-1))

Outer Loop Analysis

The outer loop's control variable is j, which is initialized to 2 and updated by j = j + j/2. This simplifies to j = (3/2) * j. Let's trace the values of j:

- Iteration 1: j = 2
- Iteration 2: j = 2 * (3/2)
- Iteration 3: j = 2 * (3/2)^2
- Iteration p: j = 2 * (3/2)^(p-1)

The outer loop terminates when j >= n. So, we need to find p such that:

2 * (3/2)^(p-1) >= n(3/2)^(p-1) >= n/2Taking log (base 3/2) on both sides:

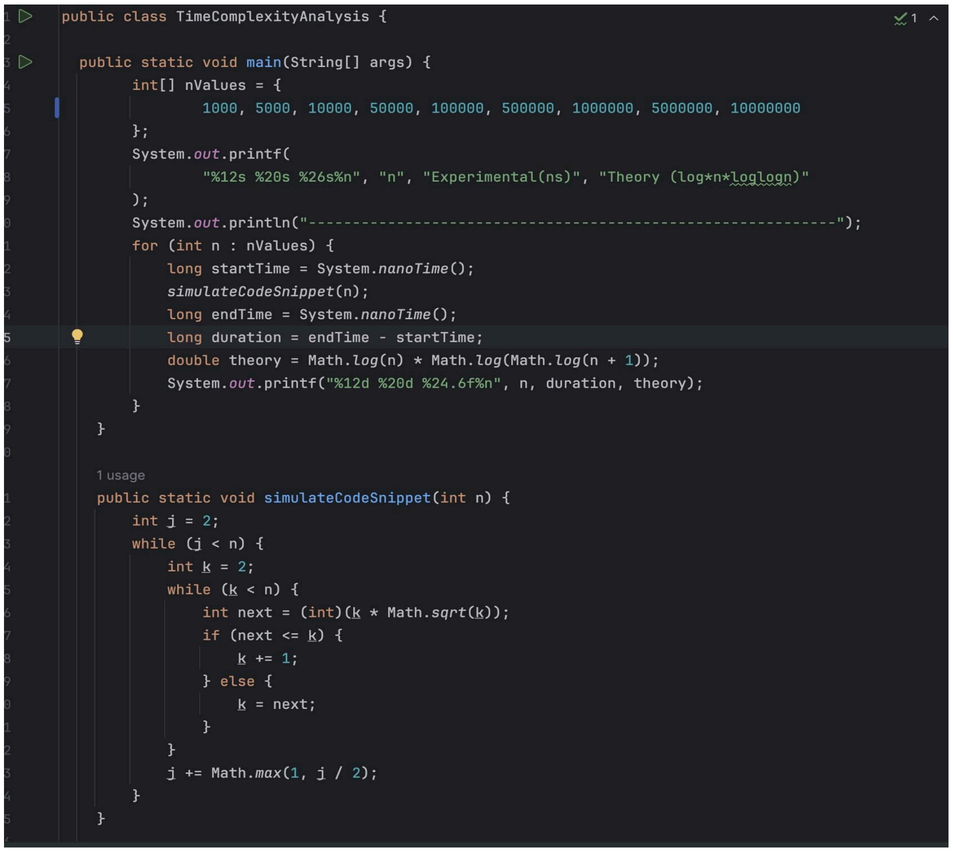
p-1 $>= \log_{3/2}(n/2)$p $>= \log_{3/2}(n/2) + 1$Therefore, the outer loop runs approximately O(log n) times.
Overall Time Complexity

Overall Time Complexity = O(log n) * O(log(log n))

Overall Time Complexity = O(log n * log(log n))

3 Experimental Analysis

3.1 Program Listing



```
public class TimeComplexityAnalysis {

  public static void main(String[] args) {
        int[] nValues = {
                1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000
        };
        System.out.printf(
                "%12s %20s %26s%n", "n", "Experimental(ns)", "Theory (log*n*loglogn)"
        );
        System.out.println("------------------------------------------------------------");
        for (int n : nValues) {
            long startTime = System.nanoTime();
            simulateCodeSnippet(n);
            long endTime = System.nanoTime();
            long duration = endTime - startTime;
            double theory = Math.log(n) * Math.log(Math.log(n + 1));
            System.out.printf("%12d %20d %24.6f%n", n, duration, theory);
        }
    }

    1 usage
    public static void simulateCodeSnippet(int n) {
        int j = 2;
        while (j < n) {
            int k = 2;
            while (k < n) {
                int next = (int)(k * Math.sqrt(k));
                if (next <= k) {
                    k += 1;
                } else {
                    k = next;
                }
            }
            j += Math.max(1, j / 2);
        }
    }
```

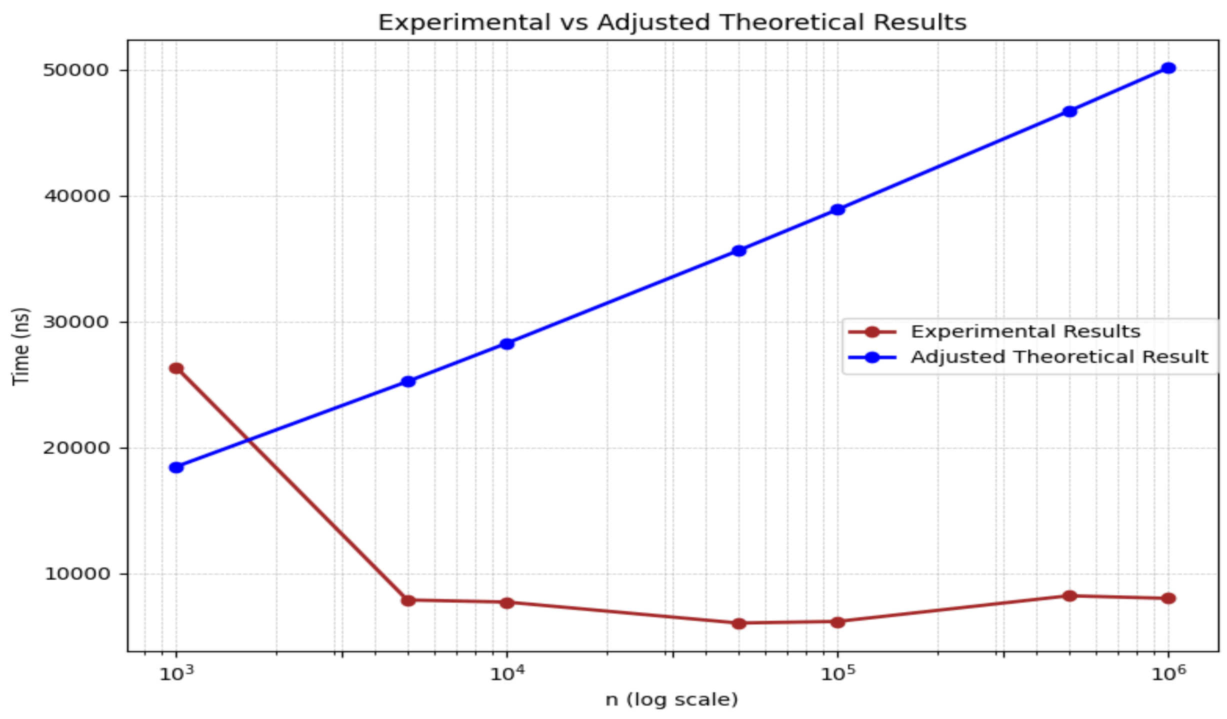
3.2 Data Normalization Notes

We normalize the theoretical values by applying a scaling constant to make them comparable with the experimental times. This scaling constant is calculated by dividing the experimental time by the theoretical value at a chosen reference input size ‘n’. The resulting constant is then used to scale all theoretical values accordingly, ensuring that both datasets can be meaningfully compared and graphed on the same scale.

3.3 Output Numerical Data

n	Experimental	Theoretical
1000	26375	13.351236
5000	7875	18.244768
10000	7708	20.450066
50000	6042	25.765978
100000	6167	28.131502
500000	8209	33.781137
1000000	8000	36.276657

3.4 Graph



3.5 Graph Observations

The plotted graph compares experimental and adjusted theoretical results for various input sizes.

1. The adjusted theoretical curve grows steadily as the input size increases.
2. Experimental timings are initially high due to overhead from the first code run and environment setup.
3. After this initial spike, experimental results drop quickly, then level off and stay consistently lower than the theory for larger inputs.
4. Both curves generally increase with input size, but the gap between them widens slightly at higher values.
5. The graph shows that practical factors affect real-world timings but the theoretical model captures the overall growth trend effectively.

4 Conclusions

From the graph above, we can conclude that as N increases exponentially, the time taken by the algorithm during the experiment aligns with the theoretical results, i.e, **O(log n * log(log n))**

5 Github Code Repository URL:

The Java code, python graph generator and README file are pushed in the below repo URL:

https://github.com/siddheshparab309/Asymptotic-Analysis/tree/main