Project 1

Aniket Konkar

1. Problem Statement

We are required to analyze the following program/code sample.

```
j = 2
while (j < n) {
        k = j
        while (k < n) {
                Sum += a[k]*b[k]
                k = k * k
        }
        j += log k
}
```

2. Theoretical Analysis

In the outer while loop, the code line `k = j` will always take O(1) time to execute, hence can be ignored. To calculate the time complexity analysis of the outer loop:

`while(j < n){ … j += log k}`

'j' starts with 2 and the loop runs until j becomes greater than or equal to n. The increment statement for this loop is *j += log k.* The value of k is manipulated by the inner while loop's statement: *k = k * k*. When the inner while loop is finished executing, the value of k is greater than or equal to n (*k >= n*). Hence, we can consider the increment as *j += log n*. Hence, the **time complexity of the outer loop is O(n/log(n)).**

For the inner while loop, the code line `Sum += a[k] * b[k]` will be always executed in constant time O(1), hence can be ignored. To Calculate the Time Complexity analysis of the inner while loop:

`while(k < n){ … k = k * k }`

'k' starts with some initial value(in this case k = j = 2) and is multiplied by itself repeatedly until it becomes greater than or equal to (>=) n. Hence, the growth of k is exponential. With initial value of k = 2, then we get following iterations k^2 = 4, (K^2)^2 = K^4 = 16, (k^4)^2 = k^8 = 256 … generalizing this, we understand that after 'm' iterations, 'k' becomes k^(2^m). At some point, k^(2^m) will become greater than or equal to n, hence we need to find how many iterations 'm' it takes for 'k' to become greater than or equal to 'n'. i.e. we need to solve the following inequality:

k^(2^m) >= n

2^m * log(k) >= log(n) … Taking log on both sides

2^m >= log(n) / log(k) … Dividing by log(k) on both sides

log(2^m) >= log(log(n) / log(k)) … Taking log on both sides (Assume log base 2)

m >= log(log(n) / log(k))

Hence, for k to become >= n, the number of iterations 'm' is about log(log n / log k). We can ignore the divide by 'log (k)' as 'k' is a constant. Hence, the **time complexity of the inner loop is O(log(log (n))).**

Here, Total Time Complexity: Outer loop runtime * Inner loop runtime = O((n/log(n)) * log(log n)) i.e.

Total Time Complexity = O((n * log(log n)) / log n)

3. Experimental Analysis - 3.1 Program Listing

```
import math
from time import perf_counter_ns
j, n = 2, 12000000; startTimer = perf_counter_ns()
while j < n: # Other Test Input Values (n): 10, 128, 1125, 16384, 107050, 1234567
    k = j
    while k < n:
        # sum += a[k] * b[k] -- This statement will always execute in O(1) time, hence commented
        k = k * k
    j += math.frexp(k)[1] - 1 # This statement is equivalent of j += log k (base 2)
endTimer = perf_counter_ns(); total_elapsed_time = endTimer - startTimer # time in nano seconds
print(total_elapsed_time) # Total elapsed time which is our experimental result
```

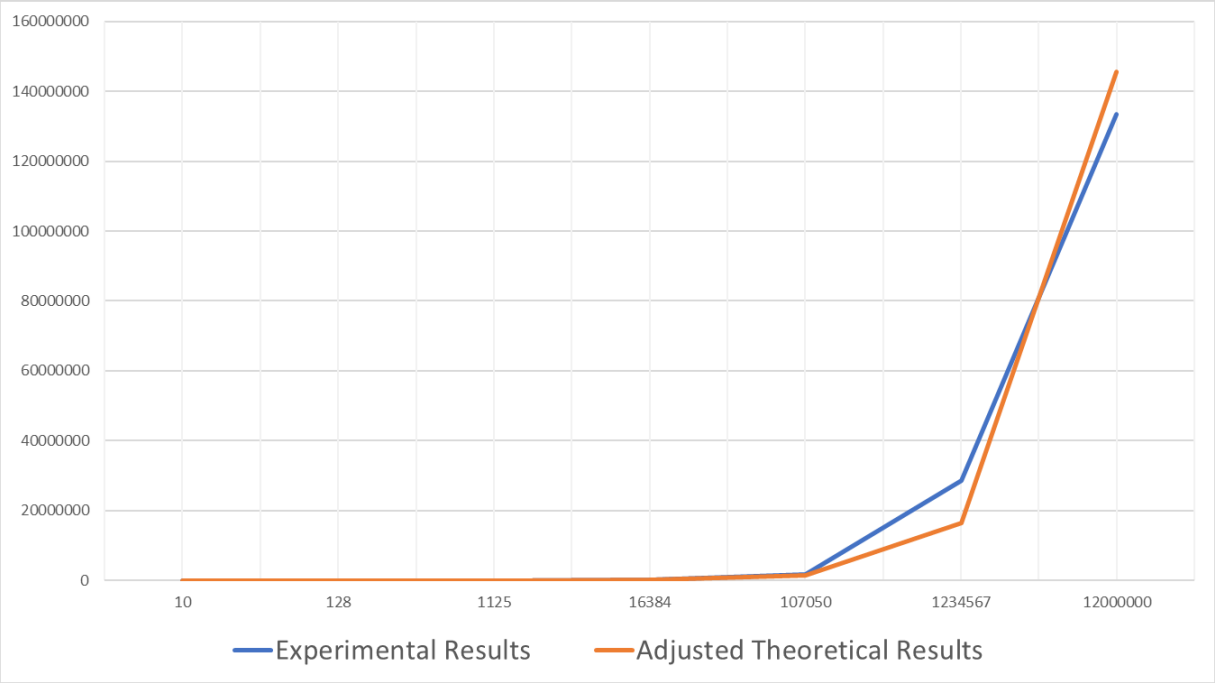
3.2 Data Normalization Notes

The Scaling Constant used here is 208.17. The formula to calculate scaling constant:
Scaling Constant = Average of Experimental Results (in ns) / Average of Theoretical Results.
Formula used to calculate Adjusted Theoretical Result:
*Adjusted Theoretical Result = Scaling Constant * Theoretical Result*

3.3 Output Numerical Data

Input Values (n)	Experimental Results (in ns)	Theoretical Results	Adjusted Theoretical Results
10	4300	1.57	326.73
128	8300	15.45	3216.93
1125	35100	111.64	23241.04
16384	237800	1341.30	279220.39
107050	1780900	7835.43	1631117.23
1234567	28686600	79685.76	16588335.77
12000000	133447700	699783.44	145675241.92
Average =>	23457242.86	112682.09	

3.4 Graph



3.5 Graph Observations

For smaller values of n, the theoretical time taken is smaller than the time portrayed by experimental results. However, for large values of n, theoretical values are greater than the experimental values as expected. Hence, the time taken for the algorithm will be no longer than the theoretical values which is the upper bound.

4. Conclusions

The time complexity for the following algorithm is **O((n * log(log n)) / log n).** Overall, it is observed that the theoretical results are similar to the experimental results.