Asymptotic Notations Week 1.2

Mayank Pratap Singh

July 2024

1 Logarithmic Properties

Understanding logarithms is crucial in computer science, especially for analyzing the time complexity of algorithms. Here are some fundamental logarithmic properties:

1. $\log_a a = 1$
2. $\log_a m^n = n \log_a m$
3. $\log_a mn = \log_a m + \log_a n$
4. $\log_a \frac{m}{n} = \log_a m - \log_a n$

Examples

- $\log_2 2^3 = 3 \log_2 2 = 3$
- $\log_3 27 = \log_3 3^3 = 3 \log_3 3 = 3$
- $\log_5 25 = \log_5 5^2 = 2 \log_5 5 = 2$

2 Complexity Analysis of a Loop

Let's analyze the complexity of the following program:

```
i = 1
while i < n:
    i = i * 2
```

Analysis

We start with $i = 1$. Let's see how i changes with each iteration when $n = 16$.

- Initial value: $i = 1 \rightarrow 2^0$
- After one iteration: $i = 2 \rightarrow 2^1$
- After two iterations: $i = 4 \rightarrow 2^2$
- After three iterations: $i = 8 \rightarrow 2^3$
- After four iterations: $i = 16 \rightarrow 2^4$

The while loop will stop when $i \geq n$. Let's assume the loop runs for k times.

$$2^k \geq n \Rightarrow 2^k = n$$

Taking the logarithm on both sides:

$$k \log_2 2 = \log_2 n \Rightarrow k = \log_2 n$$

For $n = 16$:

$$\log_2 16 = \log_2 2^4 = 4 \log_2 2 = 4$$

So, the time complexity of this code is $O(\log n)$.

3 Illustrations

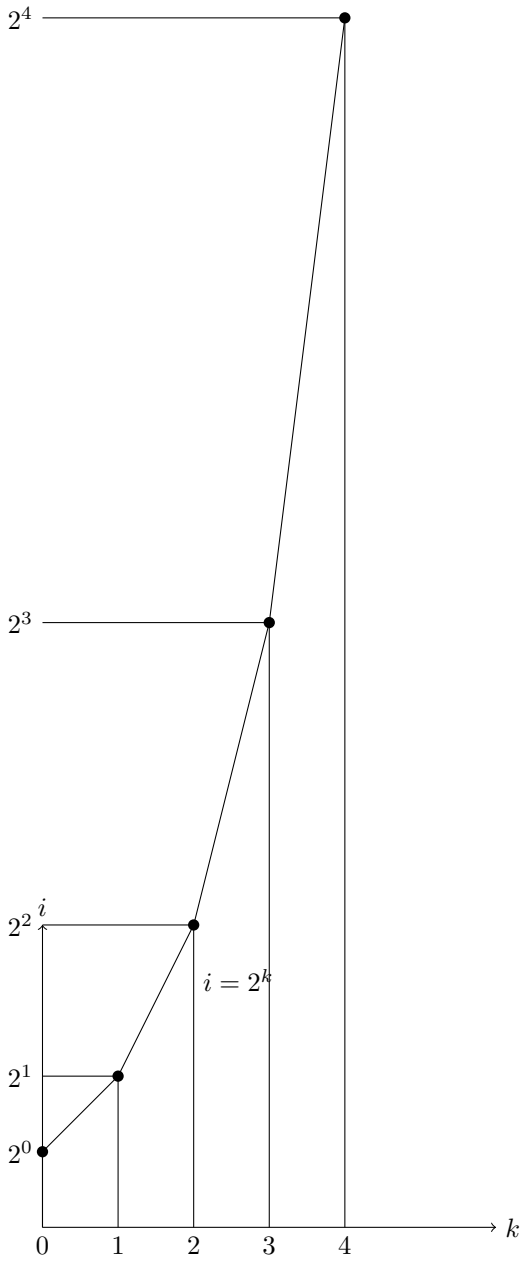

Figure 1: Exponential Growth of i in the While Loop

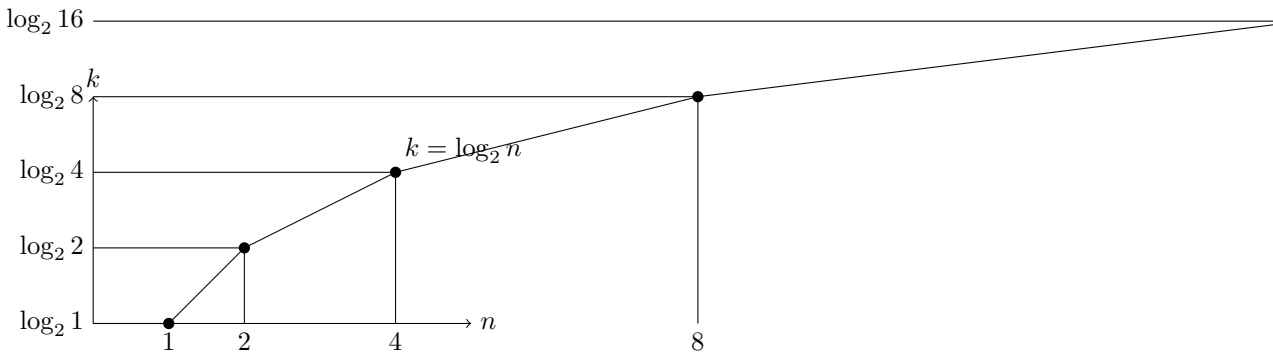

Figure 2: Logarithmic Growth of Iterations

4 Conclusion

The complexity analysis of the given loop demonstrates the significance of logarithmic properties in determining the efficiency of algorithms. By understanding how the loop variable grows exponentially and how logarithms help in analyzing this growth, we can conclude that the time complexity of this program is $O(\log n)$.

5 Complexity Analysis of a Loop

Let's analyze the complexity of the following program:

```
i = 1
while i < n:
    i = i * 10
```

Analysis

We start with $i = 1$. Let's see how i changes with each iteration when $n = 1000$.

- Initial value: $i = 1 \rightarrow 10^0$
- After one iteration: $i = 10 \rightarrow 10^1$
- After two iterations: $i = 100 \rightarrow 10^2$
- After three iterations: $i = 1000 \rightarrow 10^3$

The while loop will stop when $i \geq n$. Let's assume the loop runs for k times.

$$10^k \geq n \Rightarrow 10^k = n$$

Taking the logarithm on both sides:

$$k \log_{10} 10 = \log_{10} n \Rightarrow k = \log_{10} n$$

For $n = 1000$:

$$\log_{10} 1000 = \log_{10} 10^3 = 3 \log_{10} 10 = 3$$

So, the time complexity of this code is $O(\log n)$.

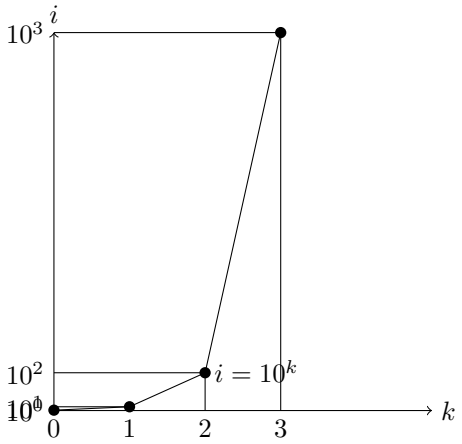

Figure 3: Exponential Growth of i in the While Loop

6 Problem 2

Consider the following program:

```
i = n
while i > 0:
    i = i - 3
```

Analysis

We start with $i = n$. The loop decrements i by 3 in each iteration until i reaches 0. Let's see how i changes with each iteration when $n = 9$.

- Initial value: $i = 9$
- After one iteration: $i = 6$
- After two iterations: $i = 3$
- After three iterations: $i = 0$

The while loop will stop when $i \leq 0$. Therefore, the loop runs $n/3$ times. The time complexity of this code is $O(n)$.

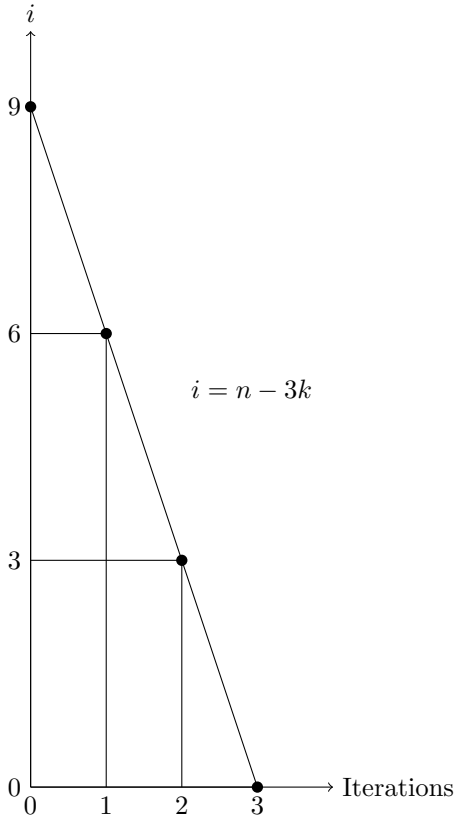

Figure 4: Linear Decrement of i by 3 in the While Loop

7 Practical Applications of Complexity Analysis

Analyzing the complexity of algorithms has several practical applications:

- **Optimization:** Helps determine if the code is optimized and efficient.
- **Algorithm Comparison:** Allows comparing different algorithms to choose the most efficient one.
- **Performance Bottlenecks:** Identifies bottlenecks in the code which can be improved.
- **Scalability:** Assesses how an algorithm scales with increasing input size.

- **Algorithm Design:** Provides insights that help in designing better algorithms by understanding the limitations of current methods.

Initial linear search algorithms had very high complexity. By analyzing and measuring the complexity, we were able to identify pain points and develop better algorithms such as binary search, which significantly improved efficiency.

Time Complexity Analysis

Let's analyze the time complexity of the given problem:

```
for i in range(n):
    k = 2
    while k < n:
        k = k * k
```

The outer loop runs n times, and the inner loop's complexity needs to be determined.

Inner Loop Analysis

Let's take $n = 64$ for understanding the inner loop's behavior.

1. Initialize $k = 2$
2. While $k < n$:
3. $k = k * k$

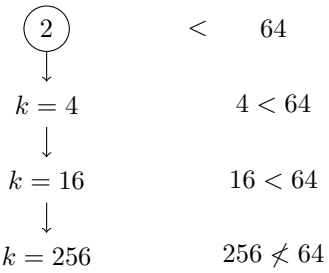

Figure 5: Steps in the inner loop for $n = 64$.

At each step, k is squared, i.e., $k = k * k$. The values of k follow the powers of 2:

$$k = 2$$
$$k = 2^2 = 4$$
$$k = 2^4 = 16$$
$$k = 2^8 = 256$$

The inner loop stops when $k \geq n$. This can be represented as $2^{2^x} \geq n$. To find x, we solve:

$$2^{2^x} = n$$

Taking log base 2 on both sides:

$$\log_2(2^{2^x}) = \log_2 n$$

$$2^x = \log_2 n$$

Taking log base 2 again:

$$x \log_2 2 = \log_2(\log_2 n)$$

$$x = \log_2(\log_2 n)$$

Thus, the inner loop runs $\log_2(\log_2 n)$ times.

Overall Time Complexity

Since the outer loop runs n times and the inner loop runs $\log_2(\log_2 n)$ times, the overall time complexity is:

$$O(n \log(\log n))$$

For large n, $\log(\log n)$ is very small, so the complexity is approximately:

$$O(n)$$

$$O(n \log(\log n)) \approx O(n)$$

Figure 6: Overall time complexity of the algorithm.

Asymptotic Notation

Asymptotic notation is used to describe the behavior of functions as inputs become large. Big O notation is commonly used to represent the worst-case scenario in terms of time complexity.

Big O Notation

Big O notation $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that:

$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

Here, $c > 0$ and $n_0 \geq 1$. This notation is used to describe the worst-case scenario.

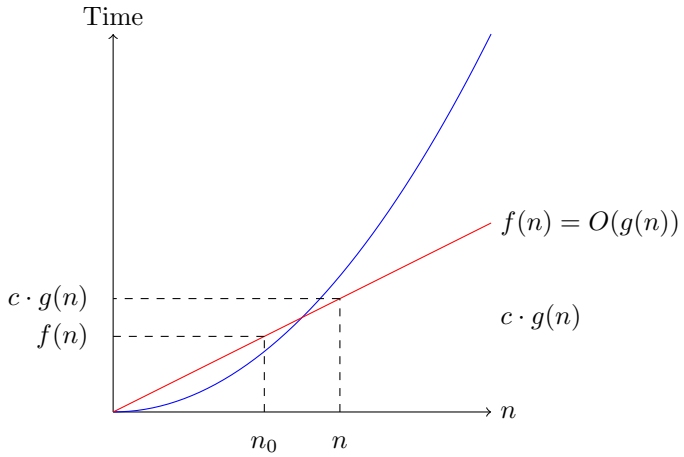

Figure 7: Graph of $f(n)$ and $c \cdot g(n)$ showing that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$.

Examples

Example 1:

$$f(n) = 5n, \quad g(n) = n$$

To show $5n = O(n)$:

$$5n \leq c \cdot n$$

Choosing $c = 5$:

$$5n \leq 5n \quad \text{(True)}$$

Thus, $5n = O(n)$.

Example 2:

$$f(n) = n, \quad g(n) = 5n$$

To show $n = O(5n)$:

$$n \leq c \cdot 5n$$

Choosing $c = \frac{1}{5}$:

$$n \leq \frac{1}{5} \cdot 5n \quad \text{(True)}$$

Thus, $n = O(5n)$.
Example 3:

$$f(n) = n^2, \quad g(n) = n$$

To show $n^2 = O(n)$:

$$n^2 \leq c \cdot n$$

There is no constant c that satisfies this for all $n \geq n_0$. Thus, $n^2 \neq O(n)$.

Common Time Complexities

- $O(1)$ - Constant time complexity (e.g., accessing an array element).
- $O(\log n)$ - Logarithmic time complexity (e.g., binary search).
- $O(n)$ - Linear time complexity (e.g., linear search).
- $O(n^2)$ - Quadratic time complexity (e.g., bubble sort).

Complexity Classes

1. $O(1)$ - Constant time complexity
2. $O(\log n)$ - Logarithmic time complexity ⟶ Binary Search
3. $O(n)$ - Linear time complexity ⟶ Linear Search
4. $O(n^2)$ - Quadratic time complexity

Figure 8: Common time complexities and their examples.

Here Linear time complexity is Linear Search and Logarithmic time compliexity is Binary Search
Understanding time complexities helps in choosing efficient algorithms. For example, binary search ($O(\log n)$) is preferred over linear search ($O(n)$) for searching in sorted arrays.

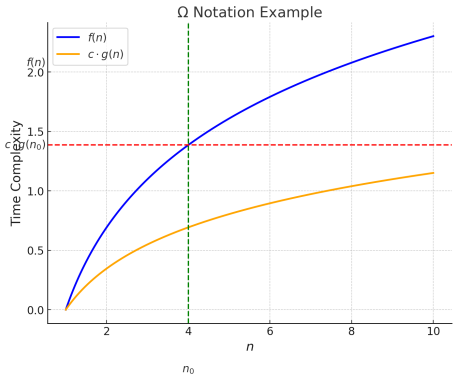

Figure 9: Graph of $f(n)$ and $c \cdot g(n)$ showing that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$.

Omega Notation

Omega notation is used to describe the best-case scenario in terms of time complexity. It is denoted by the symbol Ω.

Definition

For a given function $g(n)$, we denote $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that:

$$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

Here, $c > 0$ and $n_0 \geq 1$. This notation is used to describe the best-case scenario.

Examples

Example 1:

$$f(n) = n, \quad g(n) = 5n$$

To show $n = \Omega(5n)$:

$$n \geq c \cdot 5n$$

Choosing $c = \frac{1}{5}$:

$$n \geq \frac{1}{5} \cdot 5n \quad \text{(True)}$$

Thus, $n = \Omega(5n)$.

Example 2:

$$f(n) = 5n, \quad g(n) = n$$

To show $5n = \Omega(n)$:

$$5n \geq c \cdot n$$

Choosing $c = 5$:

$$5n \geq 5 \cdot n \quad \text{(True)}$$

Thus, $5n = \Omega(n)$.

Example 3:

$$f(n) = n, \quad g(n) = n^2$$

To show $n = \Omega(n^2)$:

$$n \geq c \cdot n^2$$

There is no constant c that satisfies this for all $n \geq n_0$. Thus, $n \neq \Omega(n^2)$.

Real-Time Example

$$\frac{n}{3} \to O(n)$$

For:

$$f(n) = \frac{n}{3}, \quad g(n) = n$$

To show $f(n) \leq c \cdot g(n)$:

$$\frac{n}{3} \leq c \cdot n$$

Choosing $c = \frac{1}{3}$:

$$\frac{n}{3} \leq \frac{1}{3} \cdot n \quad \text{(True)}$$

Thus, $\frac{n}{3} = O(n)$.

Theta Notation

Theta notation represents the average case scenario. It is denoted by the symbol Θ. In this case, both Big O and Omega equations must be satisfied.

Definition

For a given function $g(n)$, we denote $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0$$

Here, $c_1, c_2 > 0$ and $n_0 \geq 1$.

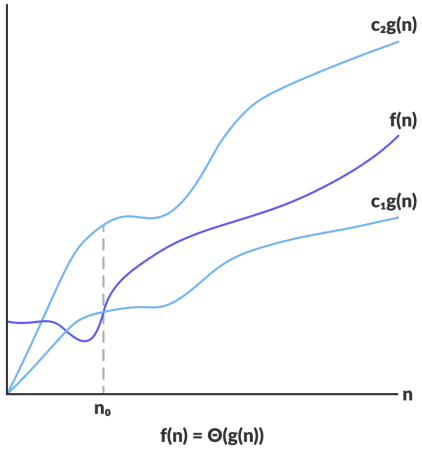

Figure 10: Graph showing $f(n) = \Theta(g(n))$.

Examples

Example 1:

$$f(n) = n, \quad g(n) = 5n$$

To show $n = \Theta(5n)$: 1. Big O:

$$n \leq c_1 \cdot 5n$$

Choosing $c_1 = \frac{1}{5}$:
2. Omega:

$$n \geq c_2 \cdot 5n$$

Choosing $c_2 = \frac{1}{5}$:
Both constants are satisfied, so $n = \Theta(5n)$.

Complexity Classes

- $O(1)$ - Constant time complexity
- $O(\log n)$ - Logarithmic time complexity (e.g., Binary Search)
- $O(n)$ - Linear time complexity (e.g., Linear Search)
- $O(n^2)$ - Quadratic time complexity
- $O(n^3)$ - Cubic time complexity
- $O(n^c)$ - Polynomial time complexity for $c \geq 3$
- $O(c^n)$ - Exponential time complexity for $c \geq 1$

Quiz Question 1

$$2^{n+5} = O(n^2) \quad \text{Yes/No}$$

The answer is Yes.

Quiz Question 2

$$\log_3 n = O(\log_2 n)$$

To show:

$$\log_3 n \leq c \cdot \log_2 n$$

Choosing $c = 1$:

Quiz Question 3

$$2^n = O(3^n)$$

To show:

$$2^n \leq c \cdot 3^n$$

Choosing $c = 1$: