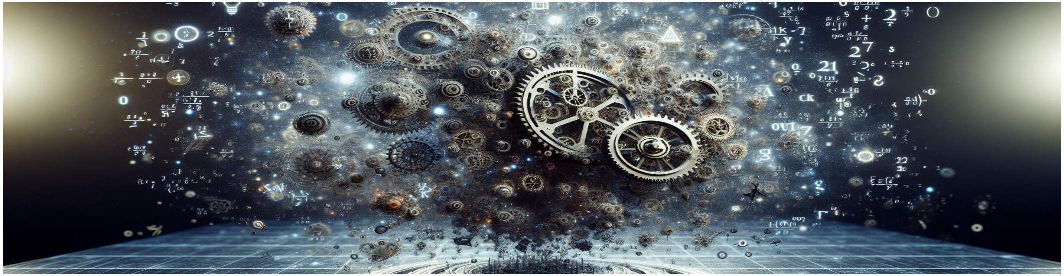


Foundations of Computer Science - Big O Notation

Sparky Training Academy
Audited by Yuri Malitsky

Big O Notation Explained

Introduction to Big O Notation

- **Definition:** Big O Notation (O) describes the maximum rate of growth of time complexity as the input size increases, emphasizing the worst-case scenario.
- **Role in CS:** Critical for evaluating how an algorithm's performance scale changes with an increasing dataset or under peak operational load.
- **Practical Example:**
  - Searching for an item in an unsorted list has time complexity $O(n)$.
  - In contrast, binary search in a sorted list has time complexity $O(\log n)$.
- **Analogy:** Shopping time increases linearly with the number of items if each item is processed individually, compared to multiple registers working concurrently.

Understanding Big O Notation

Defining Big O Notation

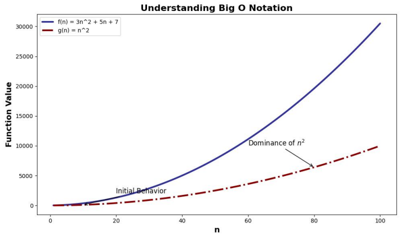
- Big O: Describes worst-case upper bound complexity.
- Example: Loop runs n times; complexity is O(n).

Importance of the Highest Order Term

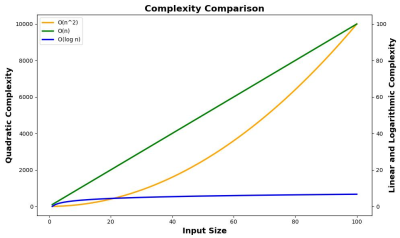
- Focus on largest growth term in Big O.
- Lower terms and constants ignored.

When analyzing complexity, consider function $f(n) = 3n^2 + 5n + 7$. In Big O notation, this simplifies to $O(n^2)$.

Explanation:
As n increases, n^2 term governs the growth, overshadowing $5n$ and 7, which become negligible.



Practical Impact of Big O



Understanding Big O Notation

- **Definition:** Quantifies worst-case scenario efficiency of an algorithm with increasing input size.
- **Algorithm Efficiency:** Essential for performance in large-scale data processing.
- **Practical Example:** Consider sorting algorithms - Selection sort typically exhibits $O(n^2)$.

Real-World Impact of Big O

- **Algorithm Efficiency Comparison:**
  - $O(n)$ vs. $O(n^2)$ sorting impact.
- **Optimized Algorithms:** Prioritize $O(n)$ for large datasets.

Time Complexities Catalog

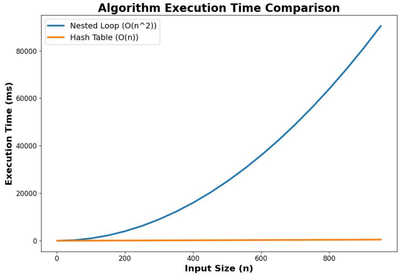
Time Complexities Catalog

Complexity Type	Characteristics
$O(1)$	Constant: Does not vary with input size
$O(n)$	Linear: Increases directly with input size
$O(n^2)$	Quadratic: Each element interacts with others
$O(2^n)$	Exponential: Doubles with each input increment
$O(\log n)$	Logarithmic: Decreases data input size stepwise

```python
# Example of Linear Time Complexity O(n)
def find_max(data):
    max_val = data[0]  # Assume non-empty list
    for num in data:
        if num > max_val:
            max_val = num
    return max_val
```

Explanation: The function `find_max` iterates through each element to find the maximum value, reflecting a linear relationship with the size of the data n.

Space vs. Time in Algorithms



Understanding Space and Time Complexity

- **Space Complexity**: Total memory usage, includes constants.
- **Time Complexity**: Worst-case step count as input size grows.

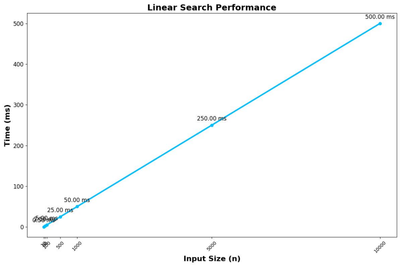
Trade-offs Between Space and Time Complexity

- **Space vs. Time Complexity:** Nested loop for finding duplicates: Space O(1), Time O(n^2).
- **Balancing Complexity:** Hash table use: Space O(n), Time O(n).
- **Practical Application:** Nested loops optimal for small, space-sensitive lists.

Calculating Big O

Calculating Big O for Linear Search

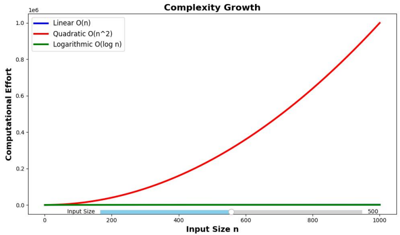
- **Big O Notation**: Measures worst-case efficiency.
- **Linear Search Steps**: Inspect each element till target found.
- **Big O Calculation**:
    - Worst case: target last or absent.
    - Inspect n elements for n-sized list.
    - Dominant term: n, thus O(n).
- **Significance**: Predict performance scalability.



Growth Rates Visualized

Growth Rates Visualized

- Interactive visual of algorithm complexity.
- Impact of input size on computational effort.
- **Example:** Graph with adjustable slider for ‘n‘ illustrating:
  - **O(n)**: Linear increase with slider adjustment.
  - **O(n^2)**: Quadratic spike, inefficiency at large ‘n‘.
  - **O(log n)**: Mild rise, efficient at scale.



Big O in the Wild

Big O in Real-World Applications

- **Critical Role in Databases:** Optimizing sorting and searching operations.
- **Sorting Algorithms in Online Searches:** Essential for quick ranking of search results.
- **Database Management:** Enhances data retrieval for rapid querying.
- **Practical Example:** Use of O(log n) to optimize sorting by last access date.

Impact on Industry Scalability and Performance

- **Algorithm Efficiency**: Enhancing operations and scalability.
- **Performance Correlation**: Link between simplicity and performance.

Practical Example: Sorting algorithms at $O(\log n)$ allow smooth handling of more queries, promoting scalability, and cost reduction.

Recap Big O Essentials

Recap: Big O Essentials

- **Definition:** Big O Notation quantifies the maximum execution time or space usage of an algorithm relative to input size, emphasizing worst-case scenarios.
- **Upper Bound:** Indicates the asymptotic upper limit, symbolizing the theoretical maximum complexity not exceeded regardless of input size. Ex: For a loop running n times, the upper bound is $O(n)$.
- **Analytical Value:** Helps identify potential inefficiencies, guiding optimization based on worst-case analyses.
- **Practical Relevance:** Important for boosting software performance, enhancing scalability, and efficient resource management, crucial in fields like software engineering and database management.

Big O Mastery Review

Big O Mastery Review

- **Identifying Big O Notation:** Recognize complexity classes for algorithm efficiency.
  - $O(n)$: Scales with data size.
  - $O(\log n)$: Reduces data subset progressively.
- **Understanding Efficiency:** Evaluate runtime and resource usage.
  - $O(1)$: Constant time, peak efficiency.
  - $O(n \log n)$: Good for sorting algorithms.
- **Applying Big O Analysis:** Tailor algorithms to data and needs.
  - Example: $O(\log n)$ for fast queries in large databases.