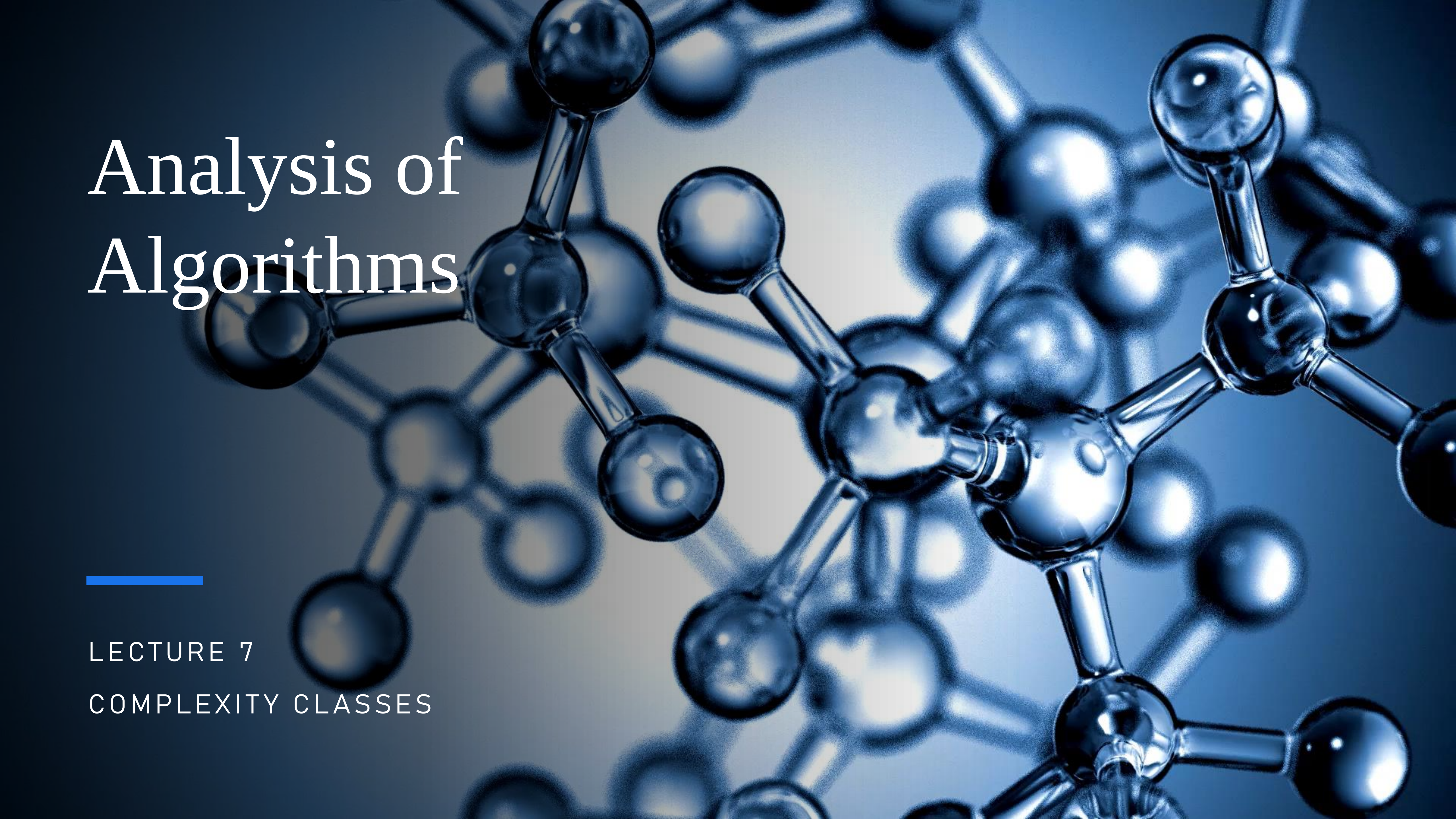


Analysis of Algorithms

LECTURE 7

COMPLEXITY CLASSES

Recap

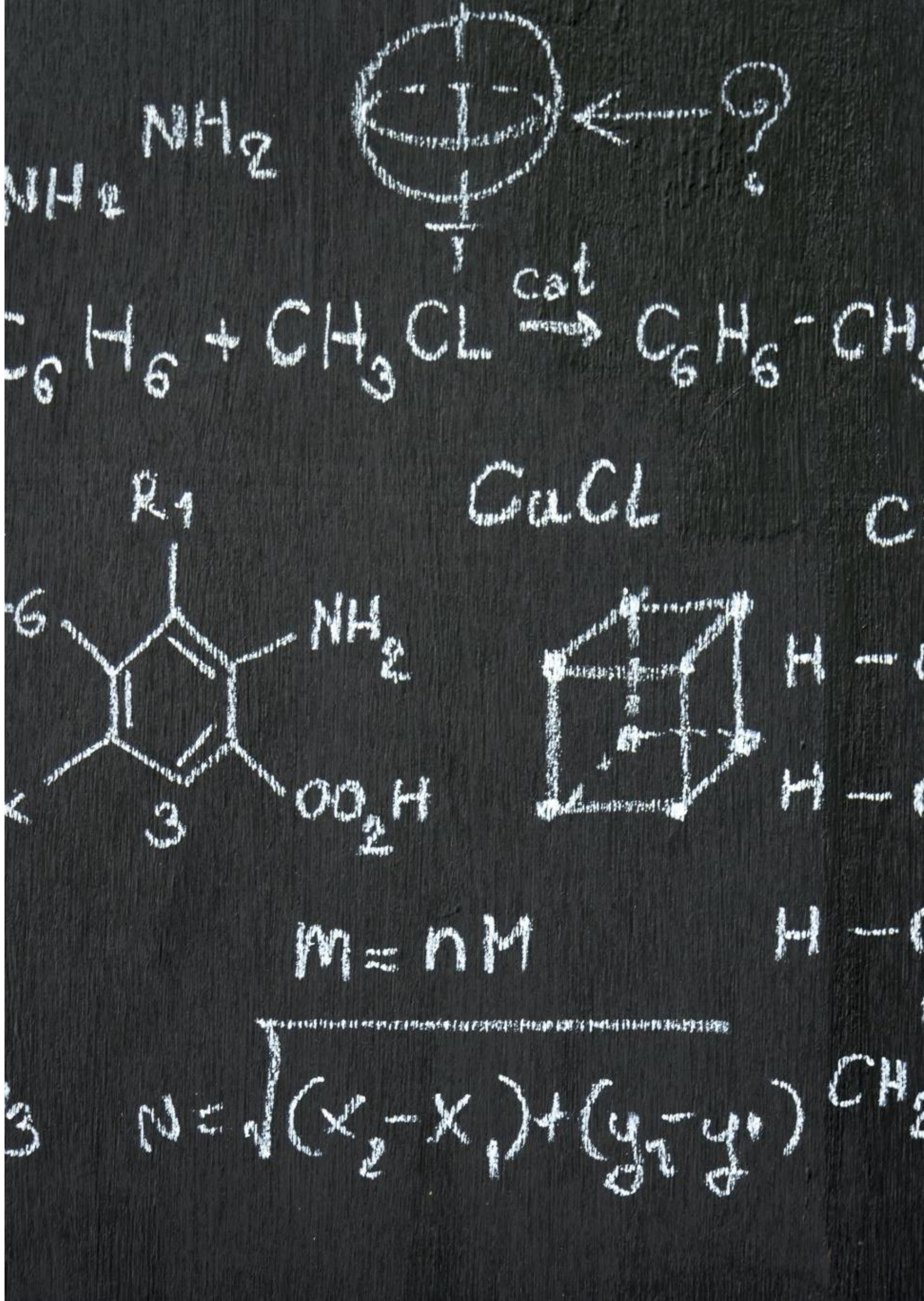
- Theta Notation (Average Case Analysis)
- Little o
- Little Omega
- Comparison

Todays Lecture

- Introduction to Complexity Classes
- Constant Time Complexity
- Linear Time Complexity
- Quadratic Time Complexity
- Log Time Complexity
- Log linear Time Complexity
- Exponential Time Complexity
- Comparison

Objective

- Compare the complexity classes Constant (O(1)), Linear (O(n)), Quadratic ($O(n^2)$), Log-linear (O(n log n)), and Exponential ($O(2^n)$) with respect to time complexity, and understand their implications in algorithm design.



Categorization into Classes

- **What is Time Complexity?**
  Time complexity measures how the runtime of an algorithm grows as the input size (n) increases.
  - It helps us predict whether an algorithm will scale well for large datasets.
- Why Study Complexity Classes?
  - Different algorithms solving the same problem can have vastly different performance.
  - Complexity informs decisions about which algorithm to use in practice (e.g., sorting 10 vs. 1 million elements).
  - Helps avoid impractical solutions for large-scale problems.
- **Big-O Notation**: Describes the **upper bound** of an algorithm's growth rate.
  - Ignores constants and lower-order terms (e.g., $O(3n^2 + 2n + 1)$ simplifies to $O(n^2)$).
  - Focuses on behavior as $n \to \infty$.

Categorization into Classes: Constant O(1)

- An algorithm has constant time complexity if its runtime does not depend on the input size.

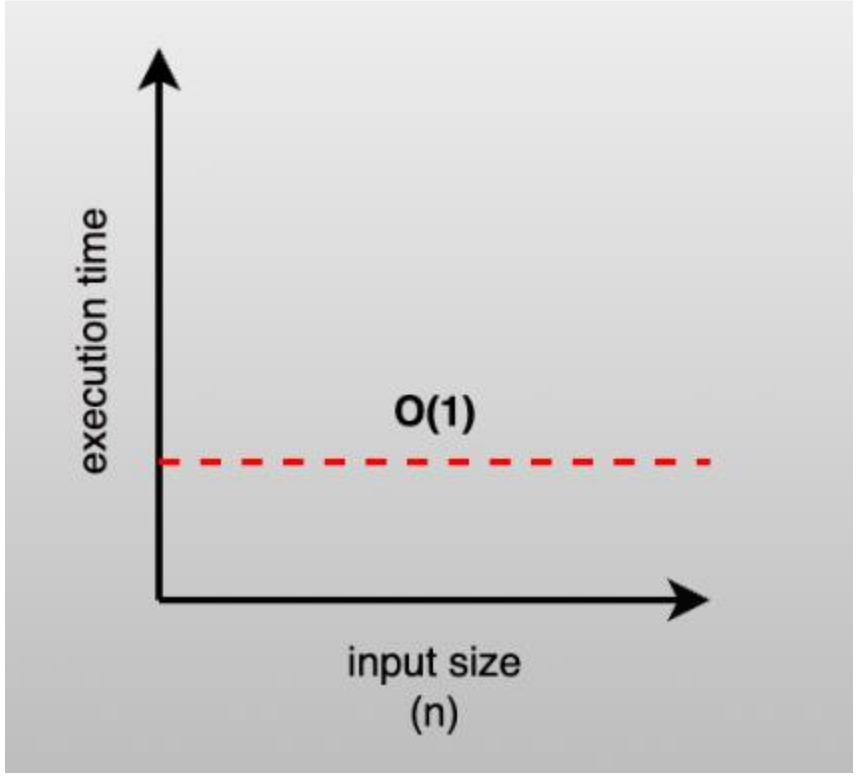
Intuition:

- No matter how big the input is, the algorithm takes the same amount of time.
- Think of it as a single, quick operation.

Examples:

1. Accessing an element in an array by index (e.g., array[5]).
2. Checking if a number is even (e.g., num % 2 == 0).
3. Hash table lookup (average case, assuming no collisions).

Categorization into Classes: Constant O(1)



Real-World Application:

- Hash tables in databases for instant key-value lookups.
- Direct memory access in low-level programming.
- if an algorithm is to return the first element of an array.

```
const firstElement = (array) => {
  return array[0];
};

let score = [12, 55, 67, 94, 22];
console.log(firstElement(score)); // 12
```

- The function above will require only one execution step, meaning the function is in constant time with time complexity O(1).

Linear Time Complexity: O(n)

An algorithm has linear time complexity if its runtime grows proportionally to the input size.

Intuition:

- If you double the input size, the runtime roughly doubles.
- Typically involves a single loop over the input.

Examples:

1. Finding the **maximum element** in an unsorted array (scan each element once).
2. Printing(**Iterate**) all elements in a linked list.
3. **Linear search** for an element in an unsorted list.

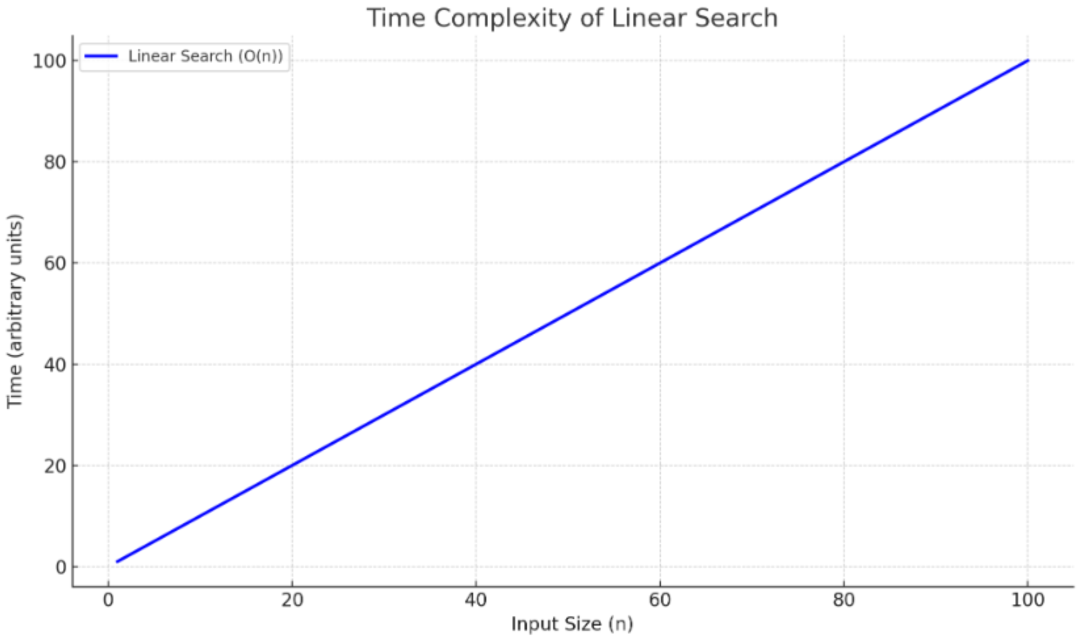
Real-World Application:

- Processing a list of user records (e.g., sending an email to each user).
- Scanning a document for a specific word.

Linear Time Complexity: O(n)

Linear Search:

```
LinearSearch(A, n, x):
    for i from 0 to n - 1 do
        if A[i] == x then
            return i       // Target found at index i
    return -1
```



Quadratic Time Complexity: O(n²)

An algorithm has quadratic time complexity if its runtime grows with the square of the input size.

Intuition:

- Often involves nested loops, where each element interacts with every other element.
- Doubling the input size quadruples the runtime.

Examples:

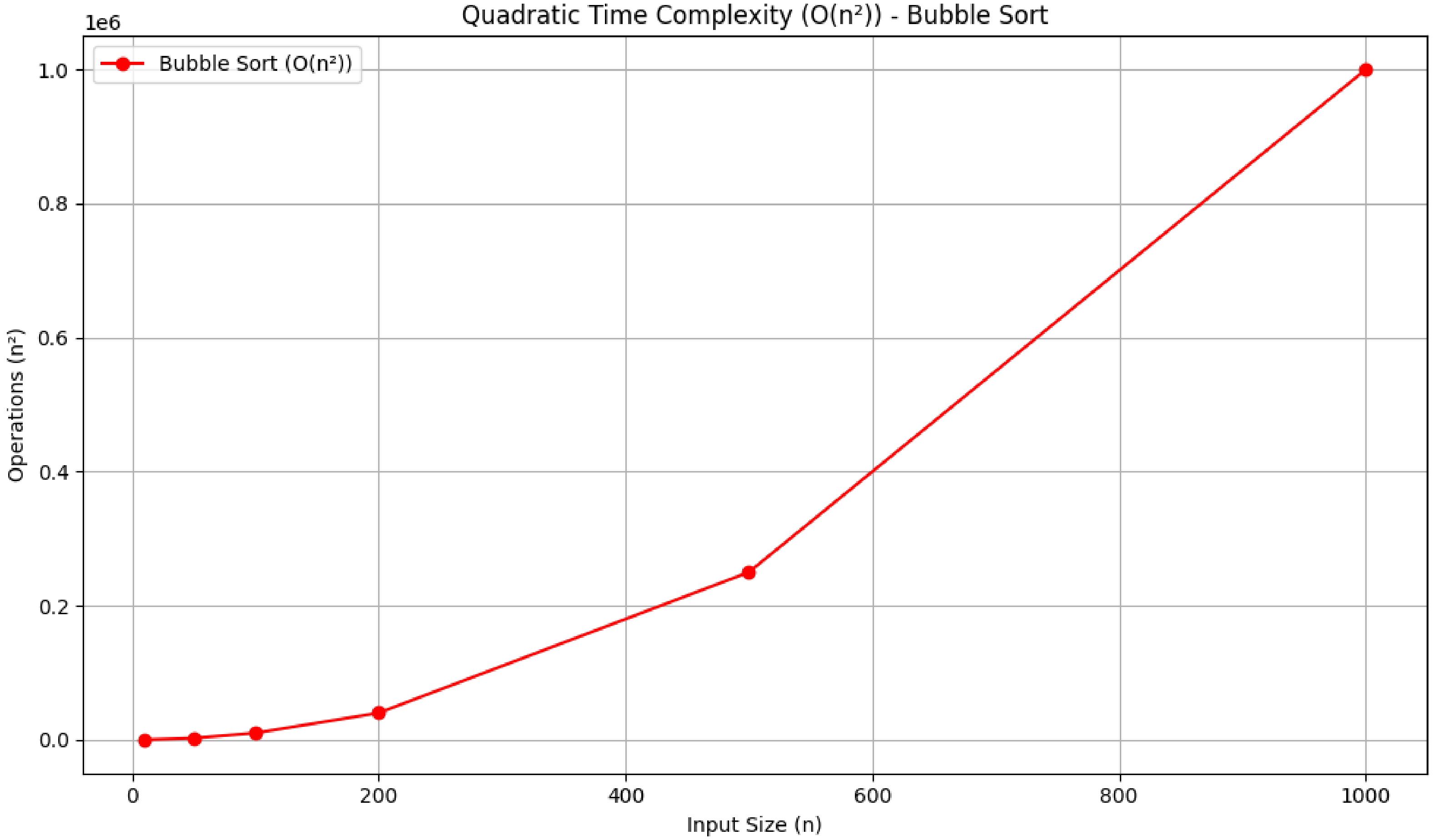
1. Bubble sort or insertion sort (comparing/swapping elements repeatedly).
2. Checking for duplicates in an unsorted array by comparing each pair.
3. Matrix multiplication (naive algorithm).

Real-World Application:

- Generating all possible pairs in a recommendation system (e.g., friend suggestions).
- Basic image processing (e.g., comparing every pixel with every other).

Quadratic Time Complexity: $O(n^2)$

Bubble Sort:



Logarithmic Time Complexity (O(log n))

An algorithm has **logarithmic time complexity**, denoted O(logn) if its runtime grows logarithmically with the input size n .

This means that as the input size increases, the number of operations increases very slowly, proportional to the logarithm of n.

Similar to linear time complexity, except that the runtime does not depend on the input size but rather on half the input size.

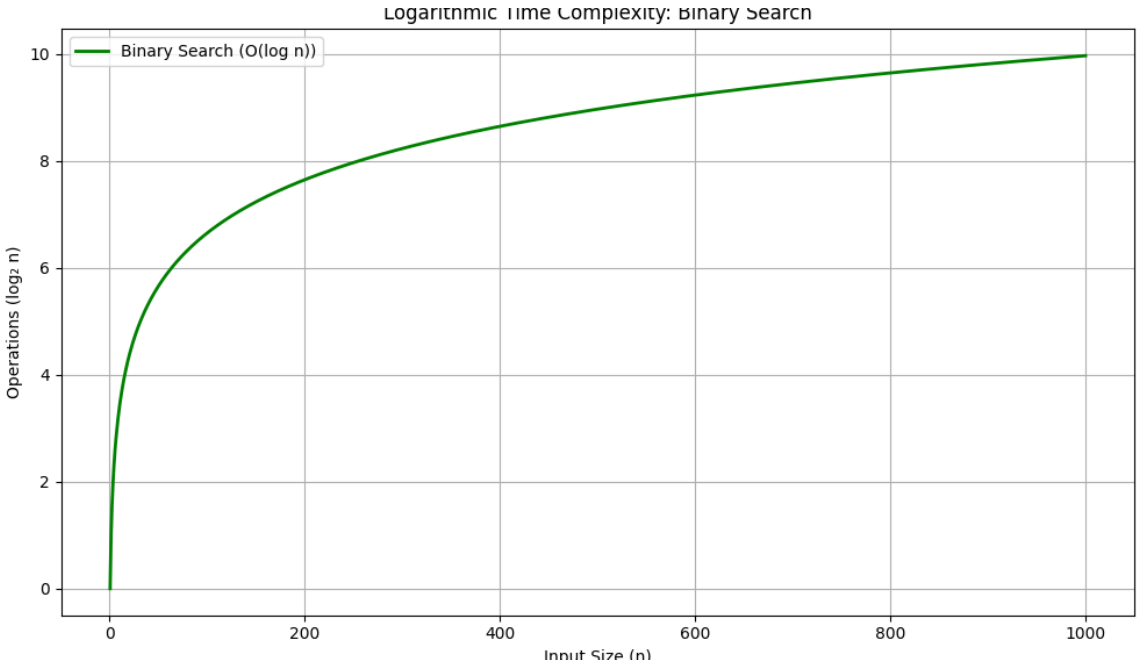
Intuition:

- Logarithmic algorithms are **highly efficient** because they typically **divide the problem size** in half (or by some constant factor) with each step.
- Think of it as: "The larger the input, the fewer additional steps need compared to linear growth."
- For example, in base-2 logarithm ($\log_2 n$), the number of steps is the number of times you can divide n by 2 until you reach 1.

If an algorithm takes k steps to process an input of size n, and $n \approx 2^k$, then $k \approx \log_2 n$.

Logarithmic Time Complexity (O(log n))

- **Binary Search**: Problem: Find an element in a sorted array.
- Approach: Repeatedly divide the search interval in half.
- Example: For n=1024, binary search takes at most $\log_2 1024=10$ steps.




```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Complexity: $O(\log n)$, since the array size is halved each iteration.

Logarithmic Linear Time Complexity (O(nlog n))

Log-linear Time Complexity (O(n log n))

- **Definition**:
  An algorithm has **log-linear time complexity**, denoted O(nlogn) if its runtime grows proportionally to the product of the input size n and the logarithm of n. This complexity is also sometimes called **linearithmic** time complexity.

. **Intuition**:

- Common in “divide-and-conquer” algorithms, where the problem is split into smaller chunks.
- Much faster than quadratic but slower than linear.
- The log n factor often comes from halving the problem size repeatedly.

Logarithmic Linear Time Complexity (O(nlog n))

Log-linear Time Complexity (O(n log n))

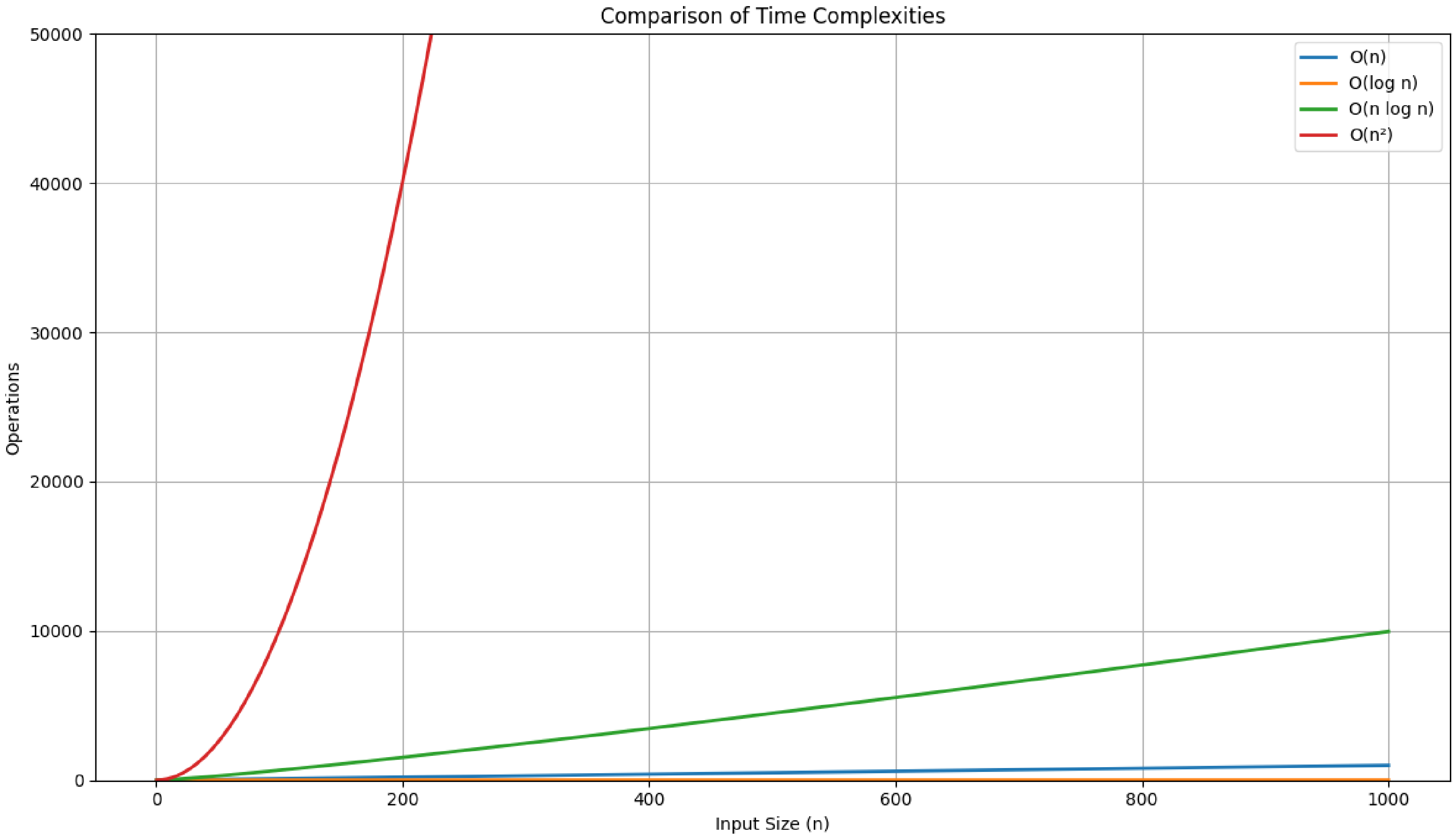
Examples:

1. Merge sort or quicksort (dividing the array and merging/sorting).
2. Fast Fourier Transform (FFT) for signal processing.
3. Heap operations (e.g., building a heap).

Real-World Application:

- Sorting large datasets (e.g., ranking search results).
- Efficient processing in computational geometry or machine learning.

Logarithmic Linear Time Complexity (O(nlog n))



Example: "For n = 1000, log n ≈ 10, so O(n log n) is ~10,000 operations, vs. O(n^2) = 1,000,000."

Exponential Time Complexity ($O(2^n)$)

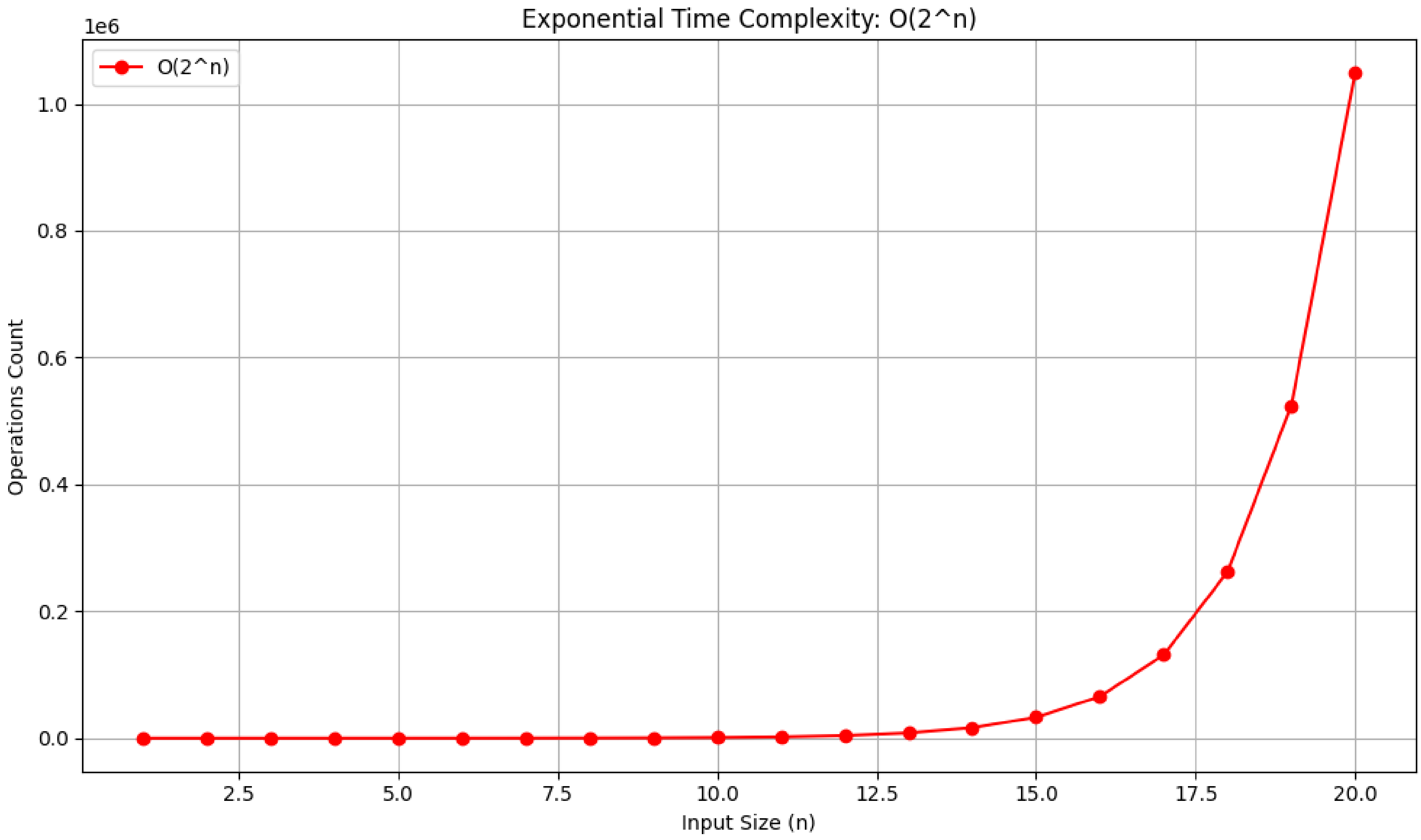
- **Definition**: An algorithm has exponential time complexity if its runtime grows exponentially with the input size, typically expressed as O(2^n), O(3^n) or generally O(k^n) k>1.

- **Characteristics**:
- The number of operations doubles (or scales by a constant factor) for each additional element in the input.
- Extremely inefficient for large inputs; runtime becomes impractical even for moderate n n n.
- **Examples**:
- Recursive algorithms solving problems like the Tower of Hanoi or generating all subsets of a set.
- Certain NP-complete problems (e.g., brute-force traveling salesman problem).

- **Intuition**: If n=10, 2^{10}=1024 operations; for n=20, $2^{20}\approx$1,000,000 operations. The growth is explosive.
- **Use Case**: Often seen in brute-force or exhaustive search algorithms.

- Algorithms with **exponential complexity** are typically avoided unless the input size is guaranteed to be small or no better solution exists.

Exponential Time Complexity (O(2n))



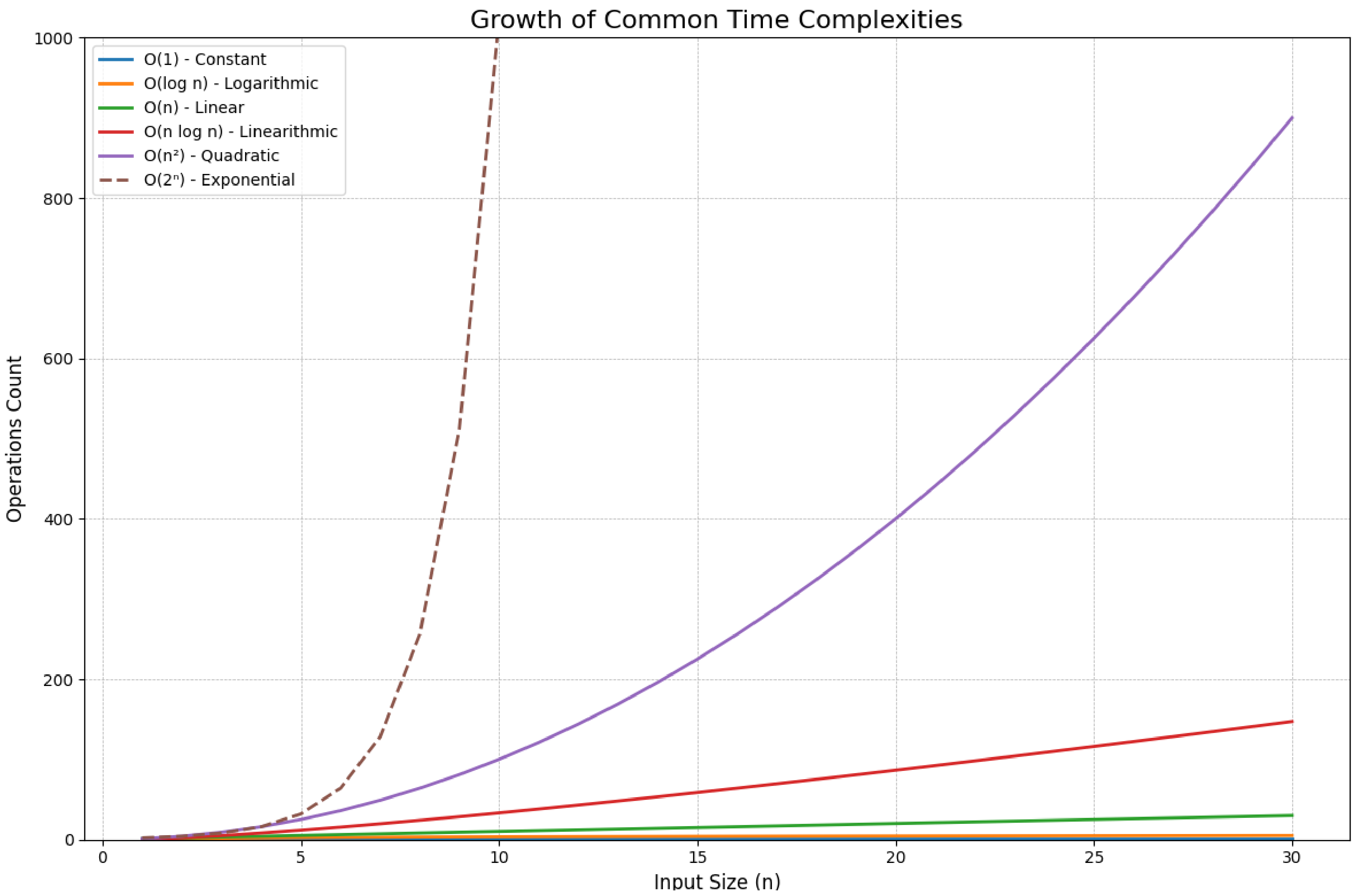
Comparison

Complexity	Big-O Notation	Growth Behavior	Example Algorithm	Operations for n=10	Operations for n=100	Use Cases
Constant	O(1)	🔶 Flat — does not grow with input	Array access, hash lookup	1	1	Caching, indexing
Logarithmic	O(log n)	🔶 Very slow growth	Binary search	~3.3	~6.6	Efficient searching
Linear	O(n)	🔷 Directly proportional	Linear search, iteration	10	100	Scanning lists, simple loops
Linearithmic	O(n log n)	🔷 Between linear & quadratic	Merge sort, quicksort avg.	~33	~664	Fast sorting
Quadratic	$O(n^2)$	⚠️ Grows fast with input size	Bubble sort, nested loops	100	10,000	Small data brute force
Exponential	$O(2^n)$	🚨 Explodes with input	Recursive Fibonacci, brute force	1024	~1.27e30	Very small input only

Comparison: Growth Visualization

Input Size (n)	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)
10	1	3.3	10	33	100	1024
20	1	4.3	20	86	400	1M
50	1	5.6	50	282	2500	1.1e15
100	1	6.6	100	664	10,000	~1e30

Comparison: Growth Visualization



Comparison: Input size tolerance

Complexity	Ideal Input Size Limit (for real-time response)
O(1), O(log n), O(n)	1 million+ items
O(n log n)	100,000 – 1 million
$O(n^2)$	Up to ~10,000
$O(2^n)$, O(n!)	20–30 max

- Best performing: O(1), O(log n)
- Balanced: O(n), O(n log n)
- Worst performing: $O(n^2)$, $O(2^n)$ (avoid for large inputs)

Analyzing Time Complexity Through Statement-Level Cost Estimation

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

Algorithm 1

	Cost
arr[0] = 0;	c_1
arr[1] = 0;	c_1
arr[2] = 0;	c_1
...	...
arr[N-1] = 0;	c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2

	Cost
for(i=0; i<N; i++)	c_2
arr[i] = 0;	c_1

$(N+1) \times c_2 + N \times c_1 =$

$(c_2 + c_1) \times N + c_2$

Another Example

- *Algorithm 3* *Cost*

```
sum = 0;                          c1
for(i=0; i<N; i++)                c2
   for(j=0; j<N; j++)             c2
         sum += arr[i][j];          c3
```

$c_1 + c_2 \, x \, (N+1) + c_2 \, x \, N \, x \, (N+1) + c_3 \, x \, N^2$

Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (approximation)

- The low order terms in a function are relatively insignificant for **large** n

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**

25

Class activity

- Analyze individual statement costs in an algorithm.
- Derive the total cost of execution.
- Identify the time complexity class (Big-O).
- Plot costs for different input sizes and recognize growth patterns

```python
N = 5
matrix = [[0 for _ in range(N)] for _ in range(N)]

for i in range(N):
    for j in range(N):
        matrix[i][j] = i * j
```

Summary

- Complexity Classes
- Comparison of Complexity classes
- Class activity

27