Notes: Comparing Orders of Magnitude & Calculating Complexity

What Does It Mean?

Comparing orders of magnitude helps us understand how algorithm performance grows with input size.

Calculating complexity means determining the time or space requirements of an algorithm using Big-O notation.

What Is an Order of Magnitude?

It refers to the scale of growth of a function.

We use it to classify algorithms based on how they grow as input size n increases.

Example: If $f(n) = n^2$ and $g(n) = n$, then f(n) grows faster than g(n).

Common Orders of Magnitude (Big-O)

O(1) - Constant (e.g., Accessing arr[0])

O(log n) - Logarithmic (e.g., Binary Search)

O(n) - Linear (e.g., Loop through array)

O(n log n)- Log-linear (e.g., Merge Sort)

$O(n^2)$ - Quadratic (e.g., Nested Loops)

$O(2^n)$ - Exponential (e.g., Recursive Fibonacci)

O(n!) - Factorial (e.g., Permutations)

Why It Matters

Orders of magnitude help compare algorithm efficiency.

Knowing which algorithm grows slower helps in choosing the best one for large inputs.

How to Calculate Time Complexity

1. Identify input size n
2. Count basic operations (comparisons, loops)

Notes: Comparing Orders of Magnitude & Calculating Complexity

3. Estimate operations as a function of n

4. Express in Big-O (drop constants/lower terms)

Example 1: Single Loop

```
def print_items(arr):
    for item in arr:
        print(item)
```

Time: O(n), Space: O(1)

Example 2: Nested Loop

```
def print_pairs(arr):
    for i in arr:
        for j in arr:
            print(i, j)
```

Time: $O(n^2)$

Example 3: Binary Search

```
def binary_search(arr, x):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
```

Notes: Comparing Orders of Magnitude & Calculating Complexity

```
    return -1
```

Time: O(log n)

Tips

- Simplify expressions: O(2n + 3) → O(n)
- Ignore constants and lower-order terms
- Focus on growth with large n

Summary

- Orders of magnitude help compare efficiency
- Use Big-O notation to calculate complexity
- Understand trade-offs for better coding decisions

Graph: Time Complexity Growth Comparison