1. Time Complexity

Time complexity is the measure of the number of operations an algorithm performs as a function of input size n.

Detailed Steps to Analyze Time Complexity

1. **Break Down the Code:**
   - Divide your program into individual operations (loops, function calls, conditions, etc.)
   - Assign complexity to each operation.
2. **Consider Iterations of Loops:**
   - Single loop over n elements → O(n)
   - Nested loops → Multiply complexities of each loop.
3. **Analyze Recursion:**
   - Determine the number of recursive calls.
   - Derive the recurrence relation and solve it (e.g., using the Master Theorem or substitution).
4. **Drop Constants and Non-Dominant Terms:**
   - Focus on the fastest-growing term as n increases.
   - Example: If complexity is $5n^2 + 3n + 7$, simplify to $O(n^2)$.

Common Scenarios and Their Complexities

Operation	Time Complexity
Accessing an array element	O(1)
Traversing a list with a loop	O(n)
Searching in a sorted array (binary search)	O(log n)
Nested loops (double traversal)	$O(n^2)$
Sorting (merge sort, quick sort)	O(nlog n)
Recursive Fibonacci calculation	$O(2^n)$

Examples

Example 1: Single Loop

```
def sum_of_elements(arr):
  total = 0
  for num in arr:
    total += num
  return total
```

- **Steps to analyze**:
  - for num in arr → Loop runs n times (where nn is the size of arr).
  - Inside loop: total += num is O(1).
- **Total Complexity**: O(n)

Example 2: Nested Loop

```
def find_duplicates(arr):
  duplicates = []
  for i in range(len(arr)):
    for j in range(i + 1, len(arr)):
      if arr[i] == arr[j]:
        duplicates.append(arr[i])
  return duplicates
```

- **Steps to analyze**:
  - Outer loop runs n times.
  - Inner loop runs n−1,n−2,…,1n-1, n-2, \dots, 1 times.
  - Total iterations: $n(n-1)/2=O(n^2)$.
- **Total Complexity**: $O(n^2)$

Example 3: Recursion

```
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n - 1)
```

- **Steps to analyze**:
  - Function calls itself n times until n = 0.
  - Each call takes O(1).
- **Total Complexity**: O(n)

Example 4: Divide-and-Conquer

```
def merge_sort(arr):
  if len(arr) <= 1:
    return arr
  mid = len(arr) // 2
  left = merge_sort(arr[:mid])
  right = merge_sort(arr[mid:])
  return merge(left, right)
```

- **Steps to analyze**:
  - Divide array into two halves → O(log n) levels of recursion.
  - Merging arrays → O(n) work at each level.
- **Total Complexity**: O(nlog n)

Advanced Time Complexity Tools

1. **Recurrence Relations** (for recursive functions):
   - Example: $T(n)=2T(n/2)+O(n)$
     - This simplifies to O(n log n) using the Master Theorem.
2. **Master Theorem**:
   - For recurrence relations of the form $T(n)=aT(n/b)+O(n^d)$:

    - a: Number of subproblems.
    - b: Factor by which input size is reduced.
    - d: Complexity of the division/merge step.
  - Use $\log_b(a)$ to compare with d.

---

2. Space Complexity

Space complexity measures the memory consumed by your program.

Detailed Steps to Analyze Space Complexity

1. **Account for Variables and Data Structures**:
   - Scalar variables (e.g., x, count) → O(1).
   - Arrays/lists → O(n), where nn is the size of the list.
   - Nested data structures (e.g., lists of lists) → Multiply dimensions.
2. **Account for Recursion**:
   - Recursive calls take stack space.
   - Depth of recursion determines space complexity.
3. **Separate Input vs. Auxiliary Space**:
   - Input space is usually not included in the calculation.

Examples

Example 1: Iterative Function

```
def sum_array(arr):
  total = 0
  for num in arr:
    total += num
  return total
```

- **Space Complexity**:
  - Variables: total → O(1)
  - Input: arrarr is not counted.

- Total: O(1)

Example 2: Recursive Function

```
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n - 1)
```

- **Space Complexity**:
  - **Base case**: When n=0, the recursion stops.
  - **Recursive depth**: The function keeps calling itself until n reaches 0.
  - For n=5:
    - Recursive calls: factorial(5),factorial(4),factorial(3),factorial(2),factorial(1)
  - Depth: 5.
- **Space Complexity of Depth**: O(n), because the call stack grows linearly with nnn.

Example 3: Data Structures

```
def reverse_array(arr):
  return arr[::-1]
```

- **Space Complexity**:
  - Input: arr is not counted.
  - Output: New array of size n →O(n).
  - Total: O(n)

Summary of Space Complexity

Scenario	Space Complexity
Single scalar variable	O(1)
Single loop (no additional array)	O(1)
Recursive function	O(depth)
Additional list of size nn	O(n)
