Function Complexity Analysis Cheat Sheet

Laio Oriel Seman - laio.seman@ufsc.br

$\forall \ \exists \in \mathbb{N} \leq \geq \sum \Theta \ \Omega \ O$

📊 Asymptotic Notation Fundamentals

Big O Notation (Upper Bound)

$f(n) = O(g(n))$ if $\exists\, c > 0, n_0$ such that $f(n) \leq c\cdot g(n)$ for all $n \geq n_0$

Mathematical Definition:

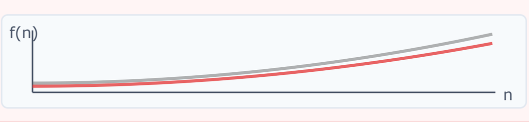
$O(g(n)) = \{f(n) : \exists\, c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq c\cdot g(n)\}$

Interpretation:

Worst-case asymptotic upper bound

Example:

$f(n) = 3n^2 + 2n + 1 = O(n^2)$



Big Theta Notation (Tight Bound)

$f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Mathematical Definition:

$\Theta(g(n)) = \{f(n) : \exists\, c_1, c_2 > 0, n_0, \forall n \geq n_0, c_1\cdot g(n) \leq f(n) \leq c_2\cdot g(n)\}$

Interpretation:

Tight asymptotic bound (exact growth rate)

Example:

$f(n) = 2n^2 + 3n = \Theta(n^2)$

Big Omega Notation (Lower Bound)

$f(n) = \Omega(g(n))$ if $\exists\, c > 0, n_0$ such that $f(n) \geq c\cdot g(n)$ for all $n \geq n_0$

Mathematical Definition:

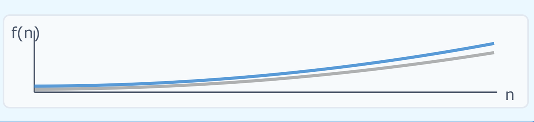
$\Omega(g(n)) = \{f(n) : \exists\, c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c\cdot g(n) \leq f(n)\}$

Interpretation:

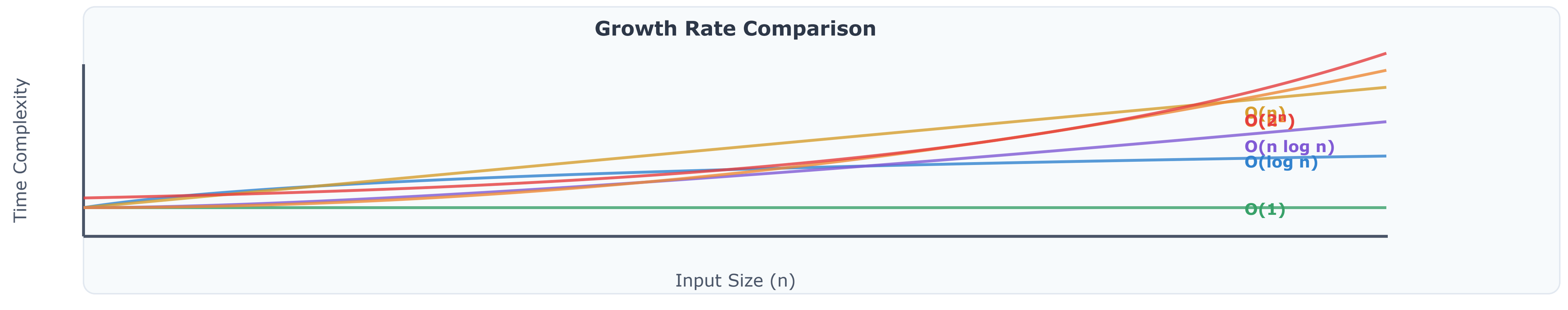
Best-case asymptotic lower bound

Example:

$f(n) = n^2 + 100n = \Omega(n^2)$



⚡ Common Complexity Classes



O(1) - Constant

Algorithm takes same time regardless of input size

Function:

$f(n) = c$

Examples:

- Array access
- Hash table lookup
- Stack push/pop

Growth: No growth

O(log n) - Logarithmic

Time increases logarithmically with input size

Function:

$f(n) = c\cdot \log n$

Examples:

- Binary search
- Balanced tree operations
- Heap insert

Growth: Very slow growth

O(n) - Linear

Time increases linearly with input size

Function:

$f(n) = c\cdot n$

Examples:

- Linear search
- Array traversal
- Finding min/max

Growth: Linear growth

O(n log n) - Linearithmic

Common in efficient sorting algorithms

Function:

$f(n) = c\cdot n\cdot \log n$

Examples:

- Merge sort
- Quick sort (average)
- Heap sort

Growth: Moderate growth

O(n²) - Quadratic

Time increases quadratically with input size

Function:

$f(n) = c\cdot n^2$

Examples:

- Bubble sort
- Selection sort
- Nested loops

Growth: Fast growth

O(n³) - Cubic

Time increases cubically with input size

Function:

$f(n) = c\cdot n^3$

Examples:

- Matrix multiplication
- Triple nested loops

Growth: Very fast growth

🔬 Analysis Techniques

Substitution Method

Guess the form of solution and prove by induction

Steps:

1. Guess the form of the solution
2. Use mathematical induction to prove
3. Verify the solution works

Example:

$T(n) = 2T(n/2) + n$

Recursion Tree Method

Visualize recursive calls as a tree structure

Steps:

1. Draw tree of recursive calls
2. Calculate cost at each level
3. Sum costs across all levels

Example:

$T(n) = 2T(n/2) + cn$

Master Theorem

Direct solution for divide-and-conquer recurrences

Recurrence Form:

$T(n) = aT(n/b) + f(n)$

Three Cases:

Case 1: $f(n) = O(n^{\log_b(a-\epsilon)}) \rightarrow T(n) = \Theta(n^{\log_b(a)})$

Case 2: $f(n) = \Theta(n^{\log_b(a)}) \rightarrow T(n) = \Theta(n^{\log_b(a)} \log n)$

Case 3: $f(n) = \Omega(n^{\log_b(a+\epsilon)}) \rightarrow T(n) = \Theta(f(n))$

Example:

$T(n) = 2T(n/2) + n \rightarrow$ Case 2 $\rightarrow T(n) = \Theta(n \log n)$

Amortized Analysis

Average time per operation over sequence of operations

Example:

Dynamic array resize: $O(1)$ amortized per insertion

Methods:

- Aggregate Method: Total cost / Number of operations
- Accounting Method: Assign credits to operations
- Potential Method: Use potential function

🔍 Detailed Algorithm Analysis

🔄 Sorting Algorithms Deep Dive

Bubble Sort

Best Case:
$O(n)$ - already sorted with early termination
Average Case:
$O(n^2)$ - random order
Worst Case:
$O(n^2)$ - reverse sorted
Space:
$O(1)$ - in-place sorting

Stable: Yes
Adaptive: Yes (with optimization)

When to use:
Educational purposes, small datasets

Merge Sort

Best Case:
$O(n \log n)$ - always divides array in half
Average Case:
$O(n \log n)$ - consistent performance
Worst Case:
$O(n \log n)$ - worst case same as best
Space:
$O(n)$ - requires additional arrays for merging

Stable: Yes
Adaptive: No

When to use:
Large datasets, stability required, consistent performance needed

Quick Sort

Best Case:
$O(n \log n)$ - balanced partitions
Average Case:
$O(n \log n)$ - random pivots
Worst Case:
$O(n^2)$ - already sorted with poor pivot
Space:
$O(\log n)$ - recursion stack in best case, $O(n)$ worst case

Stable: No (standard implementation)
Adaptive: No

When to use:
General purpose, when average case performance is sufficient

🌳 Binary Tree Operations Analysis

Search in BST

Balanced: $O(\log n)$ - height is log n
Unbalanced: $O(n)$ - degenerates to linked list
Space: $O(h)$ where h is height

Compare with root, go left or right

Insert in BST

Balanced: $O(\log n)$ - path from root to leaf
Unbalanced: $O(n)$ - worst case skewed tree
Space: $O(h)$ for recursion stack

Find correct position and insert new node

Tree Traversal (DFS)

Balanced: $O(n)$ - visit every node once
Unbalanced: $O(n)$ - still visit every node
Space: $O(h)$ - recursion stack depth

In-order, pre-order, or post-order traversal

Level Order Traversal

Balanced: $O(n)$ - visit every node
Unbalanced: $O(n)$ - visit every node
Space: $O(w)$ where w is maximum width of tree

BFS traversal using queue

🌐 Graph Algorithms Analysis

Depth-First Search (DFS)

Time Complexity:
$O(V + E)$ - visit each vertex and edge once

Space Complexity:
$O(V)$ - recursion stack or explicit stack

Use Cases:
Cycle detection, topological sorting, connected components

Implementation: Recursive or iterative with stack

Breadth-First Search (BFS)

Time Complexity:
$O(V + E)$ - visit each vertex and edge once

Space Complexity:
$O(V)$ - queue for storing vertices

Use Cases:
Shortest path in unweighted graph, level-wise traversal

Implementation: Iterative with queue

Dijkstra's Algorithm

Time Complexity:
$O((V + E) \log V)$ with binary heap

Space Complexity:
$O(V)$ - distance array and priority queue

Use Cases:
Shortest path with non-negative weights

Implementation: Priority queue (min-heap) based

🧮 Complexity Calculation Examples

Simple Loop Analysis

Code:

```
for i in range(n):
    print(i)
```

Analysis: Single loop: n iterations

Complexity: $O(n)$

Calculation: $\sum(i=1 \text{ to } n)\ 1 = n$

Code:

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

Analysis: Nested loops: n × n iterations

Complexity: $O(n^2)$

Calculation: $\sum(i=1 \text{ to } n) \sum(j=1 \text{ to } n)\ 1 = n^2$

Code:

```
for i in range(n):
    for j in range(i):
        print(i, j)
```

Analysis: Triangular loop: 0+1+2+...+(n-1)

Complexity: $O(n^2)$

Calculation: $\sum(i=1 \text{ to } n)\ i = n(n+1)/2 = O(n^2)$

Sorting Algorithm Analysis

Code:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Analysis: Outer loop: n times, Inner loop: (n-1), (n-2), ..., 1

Complexity: $O(n^2)$

Calculation: $\sum(i=1 \text{ to } n-1)\ i = (n-1)n/2 = O(n^2)$

Code:

```
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)
```

Analysis: $T(n) = 2T(n/2) + O(n)$, Master Theorem Case 2

Complexity: $O(n \log n)$

Calculation: $T(n) = 2T(n/2) + cn \rightarrow T(n) = O(n \log n)$

Code:

```
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi-1)
        quick_sort(arr, pi+1, high)
```

Analysis: Best: $T(n) = 2T(n/2) + O(n)$, Worst: $T(n) = T(n-1) + O(n)$

Complexity: Best: $O(n \log n)$, Worst: $O(n^2)$

Calculation: Best case: balanced partition, Worst: unbalanced

Binary Tree Analysis

Code:

```
def tree_height(root):
    if not root: return 0
    left_h = tree_height(root.left)
    right_h = tree_height(root.right)
    return 1 + max(left_h, right_h)
```

Analysis: Visit each node once: $T(n) = T(left) + T(right) + O(1)$

Complexity: $O(n)$

Calculation: Each node visited exactly once $\rightarrow O(n)$

Code:

```
def bst_search(root, key):
    if not root or root.val == key:
        return root
    if key < root.val:
        return bst_search(root.left, key)
    return bst_search(root.right, key)
```

Analysis: Balanced: $T(n) = T(n/2) + O(1)$, Unbalanced: $T(n) = T(n-1) + O(1)$

Complexity: Balanced: $O(\log n)$, Unbalanced: $O(n)$

Calculation: Height determines complexity: h = log n (balanced) or n (skewed)

Code:

```
def tree_traversal(root):
    if root:
        tree_traversal(root.left)
        print(root.val)  # Process
        tree_traversal(root.right)
```

Analysis: Visit every node exactly once

Complexity: $O(n)$

Calculation: $T(n) = T(left) + T(right) + O(1) = O(n)$

Divide & Conquer Analysis

Code:

```
def binary_search(arr, target, low, high):
    if low > high: return -1
    mid = (low + high) // 2
    if arr[mid] == target: return mid
    elif arr[mid] > target:
        return binary_search(arr, target, low, mid-1)
    else:
        return binary_search(arr, target, mid+1, high)
```

Analysis: Each call reduces problem size by half

Complexity: $O(\log n)$

Calculation: $T(n) = T(n/2) + O(1) \rightarrow O(\log n)$

Code:

```
def matrix_multiply(A, B):
    n = len(A)
    C = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
```

Analysis: Triple nested loops, each running n times

Complexity: $O(n^3)$

Calculation: $\sum\sum\sum\ 1 = n \times n \times n = n^3$

Code:

```
def power(base, exp):
    if exp == 0: return 1
    if exp % 2 == 0:
        half = power(base, exp//2)
        return half * half
    return base * power(base, exp-1)
```

Analysis: Each recursive call halves the exponent (even case)

Complexity: $O(\log n)$

Calculation: $T(n) = T(n/2) + O(1) \rightarrow O(\log n)$

Dynamic Programming Analysis

Code:

```
def fibonacci_dp(n):
    dp = [0] * (n+1)
    dp[1] = 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Analysis: Single loop, each iteration O(1)

Complexity: Time: $O(n)$, Space: $O(n)$

Calculation: n iterations $\times O(1) = O(n)$

Code:

```
def longest_common_subsequence(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0]*(n+1) for _ in range(m+1)]
    for i in range(1, m+1):
        for j in range(1, n+1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

Analysis: Fill m×n table, each cell computed in O(1)

Complexity: Time: $O(mn)$, Space: $O(mn)$

Calculation: $m \times n$ iterations $\times O(1) = O(mn)$

💾 Space Complexity Analysis

Auxiliary Space Analysis

Extra space used by algorithm (excluding input)

Examples:

- Merge sort: $O(n)$ auxiliary space for temporary arrays
- Quick sort: $O(\log n)$ auxiliary space for recursion stack
- Bubble sort: $O(1)$ auxiliary space
- Binary tree traversal: $O(h)$ space for recursion stack

Space-Time Tradeoffs

Using more space to reduce time complexity

Examples:

- Memoization: $O(n)$ space to reduce $O(2^n)$ to $O(n)$
- Hash tables: $O(n)$ space for $O(1)$ lookup
- Precomputation: Store results to avoid recalculation
- Dynamic programming: Trade space for avoiding recomputation

📐 Analysis Rules & Tips

Rule 1: Drop Constants

$O(2n) = O(n)$, $O(3n^2 + 5n + 1) = O(n^2)$

Example:

`for i in range(2*n): # Still O(n)`

Rule 2: Drop Lower Order Terms

Keep only the fastest growing term

Example:

$O(n^2 + n + 1) = O(n^2)$

Rule 3: Analyze Worst Case

Usually we care about worst-case scenario

Example:

Linear search: $O(n)$ even if item found early

Rule 4: Multiplication Principle

Nested operations multiply complexities

Example:

```
for i in range(n):
    for j in range(n): # O(n²)
```

Rule 5: Addition Principle

Sequential operations add complexities

Example:

$O(n) + O(m) = O(n + m)$

Rule 6: Recursive Relations

Use recurrence relations for recursive algorithms

Example:

$T(n) = 2T(n/2) + O(n) = O(n \log n)$

📋 Common Algorithm Complexities

Algorithm	Best Case	Average Case	Worst Case	Space
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Hash Table Insert	$O(1)$	$O(1)$	$O(n)$	$O(n)$
DFS/BFS	$O(V+E)$	$O(V+E)$	$O(V+E)$	$O(V)$
Dijkstra	$O(V \log V)$	$O(V \log V)$	$O(V \log V)$	$O(V)$
Matrix Multiplication	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^2)$

🎯 Master complexity analysis to write efficient algorithms!

Analyze worst case • Use proper notation • Consider space complexity • Practice with examples

$\exists$ elegant solutions $\forall$ complex problems