Big O Time Complexity Cheat Sheet

Overview

This cheat sheet provides a complete quick reference for Big O notation, time complexity analysis, and space complexity concepts covered in the **"Understanding Time Complexity: Your Secret Weapon for Coding Interviews"** tutorial.

Quick Reference Tables

Complexity Classes at a Glance

Complexity	Name	Growth	Performance	Example
O(1)	Constant	Flat	Lightning fast	Array access, Set lookup
O(log n)	Logarithmic	Minimal	Very fast	Binary search
O(n)	Linear	1:1 ratio	Good	Loop through array
O(n log n)	Linearithmic	Linear + a bit	Great	Merge sort, Quick sort
O(n²)	Quadratic	Exponential²	Slow	Nested loops, Bubble sort
O(n³)	Cubic	Exponential³	Very slow	Triple nested loops
O(2)	Exponential	Doubles	Never	Recursive fibonacci
O(n!)	Factorial	Multiplies	Don't	Permutations

Operations Per Input Size

When input size grows 10x:

Complexity	10 items	100 items	1,000 items	10,000 items
O(1)	1	1	1	1
O(log n)	3	7	10	13
O(n)	10	100	1,000	10,000
O(n log n)	30	700	10,000	130,000
O(n²)	100	10,000	1,000,000	100,000,000
O(2)	1,024	2		

Common Data Structure Operations

Data Structure	Access	Search	Insertion	Deletion	Space
Array	O(1)	O(n)	O(n)	O(n)	O(n)
Linked List	O(n)	O(n)	O(1)*	O(1)*	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table	N/A	O(1)**	O(1)**	O(1)**	O(n)
Binary Search Tree	O(log n)*	O(log n)*	O(log n)*	O(log n)*	O(n)
Heap	O(1)***	O(n)	O(log n)	O(log n)	O(n)
Graph	O(V+E)	O(V+E)	O(1)	O(V+E)	O(V+E)

With pointer available; ***Average case;*** Root access only

Sorting Algorithm Complexities

Algorithm	Best	Average	Worst	Space	Stable
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	No
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)	Yes
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)	Yes

How to Analyze Code

Step 1: Count Loops

```
// No loops = O(1)
function getFirst(arr) { return arr[0]; }

// One loop = O(n)
for (let i = 0; i < n; i++) { /* ... */ }

// Two nested loops = O(n²)
for (let i = 0; i < n; i++) {
```

```
  for (let j = 0; j < n; j++) { /* ... */ }
}

// Three nested loops = O(n³)
for (let i = 0; i < n; i++) {
  for (let j = 0; j < n; j++) {
    for (let k = 0; k < n; k++) { /* ... */ }
  }
}
```

Step 2: Identify Loop Type

```
// Decreases by half each iteration = O(log n)
for (let i = 1; i < n; i *= 2) { /* ... */ }

// Increases by a factor = O(log n)
for (let i = n; i > 1; i /= 2) { /* ... */ }

// Nested loops with different sizes = O(a × b)
for (let i = 0; i < a; i++) {
  for (let j = 0; j < b; j++) { /* ... */ }
}
```

Step 3: Drop Constants and Lower Terms

```
// O(2n) → O(n)
// O(n + m) → Keep as is (different variables)
// O(n² + n) → O(n²) (drop lower term)
// O(n²) + O(n²) → O(n²)
```

Step 4: Simplify

```
// O(n) + O(n) = O(2n) = O(n)
// O(n) × O(n) = O(n²)
// O(n log n) + O(n) = O(n log n) (drop lower term)
```

Big O Rules

Rule 1: Loops Add Together

```
for (let i = 0; i < n; i++) { }        // O(n)
for (let i = 0; i < n; i++) { }        // O(n)
// Total: O(n) + O(n) = O(2n) = O(n)
```

Rule 2: Nested Loops Multiply

```
for (let i = 0; i < n; i++) {
  for (let j = 0; j < n; j++) { }
```

```
  // Total: O(n) × O(n) = O(n²)
}
```

Rule 3: Drop Constants

```
O(10n) = O(n)           // Drop the 10
O(2n + 5) = O(n)        // Drop all constants
O(1000n²) = O(n²)       // Drop the 1000
```

Rule 4: Drop Non-Dominant Terms

```
O(n² + n) = O(n²)       // n² dominates
O(n log n + n) = O(n log n)  // n log n dominates
O(2n + n²) = O(n²)      // n² dominates
```

Rule 5: Different Variables

```
// Two loops on different arrays = O(a + b)
for (let i = 0; i < a; i++) { }
for (let j = 0; j < b; j++) { }

// Nested loops on different arrays = O(a × b)
for (let i = 0; i < a; i++) {
  for (let j = 0; j < b; j++) { }
}
```

Common Patterns

Pattern: O(1) - Constant Time

```
arr[0]                      // Direct access
arr[arr.length - 1]         // Direct calculation
map.get(key)                // Hash table lookup
set.has(value)              // Set lookup
obj.property                // Object property access
```

Pattern: O(log n) - Binary Search

```
// Halves the search space each iteration
while (left <= right) {
  mid = (left + right) / 2
  // Narrow search space by half
}
```

Pattern: O(n) - Linear Search

```
// Single pass through data
for (let item of arr) { }
```

```
arr.forEach(item => { })
arr.map(item => { })
arr.filter(item => { })
```

Pattern: O(n log n) - Divide and Conquer

```
// Divides: O(log n) levels
// Conquers: O(n) work per level
// Total: O(n × log n)

// Examples: Merge Sort, Quick Sort
```

Pattern: O(n²) - Nested Loops

```
for (let i = 0; i < n; i++) {
  for (let j = 0; j < n; j++) {
    // O(n²) work
  }
}
```

Pattern: O(n³) - Triple Nested Loops

```
for (let i = 0; i < n; i++) {
  for (let j = 0; j < n; j++) {
    for (let k = 0; k < n; k++) {
      // O(n³) work
    }
  }
}
```

Pattern: O(2) - Recursion Without Memoization

```
// Each call spawns 2 more calls
function fib(n) {
  if (n <= 1) return n
  return fib(n-1) + fib(n-2)  // 2 recursive calls
}
```

Space Complexity Quick Reference

Pattern	Space
Single variable	O(1)
Array of size n	O(n)
2D Array (n×n)	O(n²)
Recursion depth n	O(n)
Call stack n levels	O(n)

Pattern	Space

Interview Questions Checklist

Always discuss: - [ ] Time complexity (worst case) - [ ] Space complexity - [ ] Best/average/worst cases - [ ] Trade-offs (time vs space) - [ ] Can it be optimized?

Mention optimization if applicable: - [ ] Use better data structure - [ ] Add memoization/caching - [ ] Early termination - [ ] Preprocessing/sorting first

Be ready to answer: - "Can you optimize this?" - "What's the space complexity?" - "How does it scale?" - "Why this approach over that?"

Red Flags to Avoid

Don't do this in interviews: - Forget to discuss complexity - Confuse O(n) with $O(n^2)$ - Ignore hidden complexity - Use nested array methods ($O(n^2)$!) - Forget space complexity

Watch out for hidden complexity:

```
//  This is O(n²), not O(n)!
for (let i = 0; i < arr.length; i++) {
  if (arr.includes(arr[i])) { }  // includes() is O(n)!
}

//  This is O(n)
const seen = new Set()
for (let item of arr) {
  if (seen.has(item)) { }  // has() is O(1)
  seen.add(item)
}
```

Quick Tips for Interviews

1. **State complexity clearly**: "This is O(n log n) time, O(n) space"
2. **Explain why**: "We use a Set for O(1) lookups instead of array which is O(n)"
3. **Mention trade-offs**: "We use extra space for faster lookups"
4. **Discuss edge cases**: "Worst case is $O(n^2)$ when array is already sorted"
5. **Ask about constraints**: "How large is the input?" (might make $O(n^2)$ acceptable)

Decision Tree: Choosing an Algorithm

```
Is the array sorted?
```

```
YES
   Can we use binary search? (O(log n))
      YES → Use binary search
      NO → Use other techniques
NO
   Do we need to sort it?
      YES → Use merge sort (O(n log n))
      NO → Continue
   Can we solve with O(n) space?
      YES → Use hash map/set
      NO → Use nested loops (O(n²))
```

Performance Benchmarks (1,000,000 items)

Complexity	Time	Performance
O(1)	$<$ 1 ms	Instant
O(log n)	~20 ms	Fast
O(n)	~10 ms	Good
O(n log n)	~200 ms	Great
O(n^2)	1,000+ seconds	Too slow
O(2)	Forever	Impossible

Resources from Tutorial

Concepts covered: - What Big O notation means - Why it matters for interviews - Complexity classes: O(1) through O(2) - How to analyze code - Space complexity - Time-space trade-offs - Interview strategies

Practice problems: - Challenge 1: Analyze code complexity - Challenge 2: Optimize O(n^2) to O(n) - Challenge 3: Compare algorithms - Real-world example: Performance analyzer

Key takeaways: 1. Count loops to determine complexity 2. Nested loops multiply, sequential loops add 3. Drop constants and lower terms 4. Different variables need separate notation 5. Always consider both time and space

Common Mistakes

Mistake 1: Confusing O(n) with O(n^2)

```javascript
//  WRONG - This is O(n), not O(n²)
arr.forEach(item => console.log(item))
arr.forEach(item => console.log(item))

//  CORRECT - This is O(n²)
arr.forEach(item => {
```

```
  arr.forEach(item2 => console.log(item, item2))
})
```

Mistake 2: Ignoring Hidden Complexity

```
//  This is O(n²)
arr.forEach(item => arr.includes(item))

//  This is O(n)
const set = new Set(arr)
arr.forEach(item => set.has(item))
```

Mistake 3: Not Considering Different Cases

```
// Quick sort is O(n log n) average
// But O(n²) worst case (sorted array)
// Always mention both!
```

Next Steps

1. **Practice analyzing code** - Look at code snippets and identify complexity
2. **Solve algorithm problems** - Apply analysis to LeetCode/HackerRank problems
3. **Study optimization** - Learn when and how to optimize
4. **Mock interviews** - Practice explaining complexity clearly
5. **Teach others** - Explain Big O to solidify your understanding

---

Remember: Big O is about understanding how algorithms scale, not exact runtime. Master this and you've got the secret weapon for coding interviews!

Print this cheat sheet and reference it while practicing coding problems!