# Big-O Notation Study Guide

## 1. Key Definitions & Asymptotic Notation

### Big-O Notation (Upper Bound)

Big-O notation describes the worst-case upper bound on the growth rate of a function.

Mathematically:

T(n) is O(f(n)) ⟺ ∃ positive constants c, n₀ such that 0 ≤ T(n) ≤ cf(n), ∀ n ≥ n₀

Example: If T(n) = 3n² + 5n + 20, then T(n) = O(n²) (ignore constants and lower-order terms).

### Big-Omega (Lower Bound)

Big-Omega notation gives the lower bound, ensuring that an algorithm takes at least a certain amount of time.

Mathematically:

T(n) is Ω(f(n)) ⟺ ∃ positive constants c, n₀ such that 0 ≤ cf(n) ≤ T(n), ∀ n ≥ n₀

Example: T(n) = 4n² + 3n + 2 → T(n) = Ω(n²)

### Big-Theta (Tight Bound)

Big-Theta describes functions that have both upper and lower bounds.

Mathematically:

T(n) is Θ(f(n)) ⟺ T(n) is O(f(n)) and T(n) is Ω(f(n))

Example: T(n) = 2n² + 3n + 5 is Θ(n²) because it is both O(n²) and Ω(n²).

## 2. Common Complexity Classes & Growth Rates

|  |  |  |
| --- | --- | --- |
| Complexity | Common Example | Growth Rate |
| O(1) | Array access arr[i] | Constant |
| O(log n) | Binary search | Logarithmic |
| O(n) | Linear search | Linear |
| O(n log n) | Merge Sort, Quick Sort (avg) | Log-Linear |
| O(n²) | Bubble Sort, Selection Sort | Quadratic |
| O(2ⁿ) | Recursive Fibonacci | Exponential |
| O(n!) | Brute-force permutation | Factorial |

## 3. Mathematical Formulas for Big-O Computations

Here are common summation formulas used in analyzing complexity:

1. Arithmetic Sum Formula: ∑(i=1 to n) i = n(n+1)/2 = O(n²)

2. Geometric Sum Formula: ∑(i=0 to n) c^i = (c^(n+1) - 1) / (c - 1), for c ≠ 1

3. Summing Squares: ∑(i=1 to n) i² = n(n+1)(2n+1)/6 = O(n³)

## 4. Sorting and Searching Complexity

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best Case | Worst Case | Average Case |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) |
| Quick Sort | O(n log n) | O(n²) | O(n log n) |

## 5. Brute-force vs. Optimized Algorithms (2-Sum & 3-Sum)

Problem: Given an array and a target sum, find two numbers that add up to the target.

### Brute-Force Approach (O(n²))

Uses nested loops, checking every pair.

bool TwoSumBrute(int arr[], int n, int target) {  
 for (int i = 0; i < n - 1; i++) {  
 for (int j = i + 1; j < n; j++) {  
 if (arr[i] + arr[j] == target)  
 return true;  
 }  
 }  
 return false;  
}

### Optimized Approach (Sorting + Two Pointers) (O(n log n))

Sort the array and use two pointers.

bool TwoSumOptimized(int arr[], int n, int target) {  
 sort(arr, arr + n);  
 int left = 0, right = n - 1;  
 while (left < right) {  
 int sum = arr[left] + arr[right];  
 if (sum == target)  
 return true;  
 else if (sum < target)  
 left++;  
 else  
 right--;  
 }  
 return false;  
}

## 6. Practice Problems

1. Classify T(n) = 5n³ + 4n² + 2n + 1 in terms of Big-O, Omega, and Theta.

2. Find the complexity of this loop:

for (int i = 1; i < n; i \*= 2) {  
 cout << i;  
}

3. What is the time complexity of QuickSort in the worst case?

4. Write a function that implements binary search in O(log n) time.

5. Analyze the number of operations required for matrix multiplication (n × n).

## Answer Key

1. T(n) = O(n³), Ω(n³), Θ(n³)

2. O(log n) (doubles `i` each step).

3. O(n²) (when pivot is worst choice).

4. Binary search implementation:

int binarySearch(int arr[], int left, int right, int key) {  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 if (arr[mid] == key) return mid;  
 else if (arr[mid] < key) left = mid + 1;  
 else right = mid - 1;  
 }  
 return -1;  
}

5. O(n³).