Complete Guide to Time and Space Complexity Analysis

Table of Contents

1. Introduction to Complexity Analysis
2. Big O Notation Fundamentals
3. Time Complexity Analysis
4. Space Complexity Analysis
5. Common Complexity Classes
6. Analyzing Loops and Iterations
7. Recursion Complexity Analysis
8. Data Structure Complexities
9. Algorithm Complexity Examples
10. Best, Average, and Worst Case
11. Amortized Analysis
12. Common Mistakes and Pitfalls
13. Practice Problems
14. Master Method for Divide and Conquer

Introduction to Complexity Analysis

What is Complexity Analysis?

Complexity analysis is the process of determining how much time and space an algorithm requires as the input size grows. It helps us:

- Compare different algorithms
- Predict performance on large inputs
- Make informed decisions about algorithm choice
- Optimize code effectively

Why Does It Matter?

csharp

```
/* Example: Two ways to check if array contains duplicates */

/* Method 1: O(n²) - Nested loops */
public static bool HasDuplicates1(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)
    {
        for (int j = i + 1; j < arr.Length; j++)
        {
            if (arr[i] == arr[j])
                return true;
        }
    }
    return false;
}

/* Method 2: O(n) - Using HashSet */
public static bool HasDuplicates2(int[] arr)
{
    HashSet<int> seen = new HashSet<int>();
    foreach (int num in arr)
    {
        if (seen.Contains(num))
            return true;
        seen.Add(num);
    }
    return false;
}

/* Performance difference:
   For 10,000 elements:
   Method 1: ~50,000,000 operations
   Method 2: ~10,000 operations */
```

Big O Notation Fundamentals

What is Big O?

Big O notation describes the **upper bound** of an algorithm's growth rate. It tells us the worst-case scenario of how the algorithm will perform as input size approaches infinity.

Mathematical Definition

f(n) = O(g(n)) means there exist positive constants c and n_0 such that: **f(n) ≤ c × g(n)** for all n ≥ n_0

Simple Translation: "f(n) grows no faster than g(n)"

Key Properties of Big O

1. Constants Don't Matter

csharp

```csharp
/* These are all O(n) */
void Example1(int[] arr)
{
    foreach (int x in arr) /* n operations */
        Console.WriteLine(x);
}

void Example2(int[] arr)
{
    foreach (int x in arr)     /* n operations */
        Console.WriteLine(x);
    foreach (int x in arr)     /* n operations */
        Console.WriteLine(x * 2);
    /* Total: 2n = O(n) */
}

void Example3(int[] arr)
{
    for (int i = 0; i < arr.Length; i += 2) /* n/2 operations */
        Console.WriteLine(arr[i]);
    /* Total: n/2 = O(n) */
}
```

2. Lower Order Terms Don't Matter

csharp

```csharp
/* All of these are O(n²) */
void Example1(int[] arr) /* n² + n + 1 = O(n²) */
{
    for (int i = 0; i < arr.Length; i++)      /* n iterations */
    {
        for (int j = 0; j < arr.Length; j++)  /* n iterations each */
        {
            Console.WriteLine($"{arr[i]}, {arr[j]}"); /* n² total */
        }
    }

    foreach (int x in arr) /* +n operations */
        Console.WriteLine(x);

    Console.WriteLine("Done"); /* +1 operation */
}
```

3. Different Variables Mean Different Complexities

csharp

```csharp
/* This is O(a × b), NOT O(n²) */
void PrintPairs(int[] arrA, int[] arrB)
{
    foreach (int a in arrA)     /* a iterations */
    {
        foreach (int b in arrB) /* b iterations each */
        {
            Console.WriteLine($"{a}, {b}");
        }
    }
}

/* This is O(a + b), NOT O(n) */
void PrintArrays(int[] arrA, int[] arrB)
{
    foreach (int a in arrA)     /* a iterations */
        Console.WriteLine(a);

    foreach (int b in arrB)     /* b iterations */
        Console.WriteLine(b);
}
```

Time Complexity Analysis

Step-by-Step Process

Step 1: Identify Basic Operations

Look for the most fundamental operations that get executed:

- Arithmetic operations (+, -, *, /)
- Comparisons (<, >, ==)
- Array/list access
- Method calls

Step 2: Count How Many Times Each Operation Executes

Step 3: Express in Terms of Input Size

Step 4: Apply Big O Rules

Example Analysis

```csharp
```

```
public static int FindMax(int[] arr)
{
    int max = arr[0];            /* 1 operation */

    for (int i = 1; i < arr.Length; i++)  /* Loop runs (n-1) times */
    {
        if (arr[i] > max)        /* 1 comparison per iteration */
        {
            max = arr[i];        /* 1 assignment (worst case) */
        }
    }

    return max;                  /* 1 operation */
}

/* Analysis:
   - Initial assignment: 1
   - Loop iterations: (n-1)
   - Comparisons: (n-1)
   - Assignments: (n-1) worst case
   - Return: 1
   Total: 1 + (n-1) + (n-1) + (n-1) + 1 = 3n - 1
   Big O: O(n) */
```

Space Complexity Analysis

What is Space Complexity?

Space complexity measures the amount of extra memory an algorithm uses relative to the input size.

Types of Space Usage

1. Auxiliary Space

Extra space used by the algorithm (not counting input)

2. Total Space

Input space + auxiliary space

Note: Usually we analyze auxiliary space complexity.

Space Complexity Examples

csharp

```
/* O(1) Space - Constant extra space */
public static void ReverseArray(int[] arr)
{
    int start = 0;          /* O(1) space */
    int end = arr.Length - 1; /* O(1) space */

    while (start < end)
    {
        int temp = arr[start]; /* O(1) space */
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
    /* Total auxiliary space: O(1) */
}

/* O(n) Space - Linear extra space */
public static int[] CreateReversedArray(int[] arr)
{
    int[] reversed = new int[arr.Length]; /* O(n) space */

    for (int i = 0; i < arr.Length; i++)
    {
        reversed[i] = arr[arr.Length - 1 - i];
    }

    return reversed;
    /* Total auxiliary space: O(n) */
}

/* O(n^2) Space - Quadratic extra space */
public static int[,] CreateMultiplicationTable(int n)
{
    int[,] table = new int[n, n]; /* O(n^2) space */

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            table[i, j] = (i + 1) * (j + 1);
        }
    }
```

```
    return table;
    /* Total auxiliary space: O(n^2) */
}
```

Common Complexity Classes

Visual Representation

Performance from Best to Worst:
$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

For n = 1000:
$O(1)$: 1 operation
$O(\log n)$: ~10 operations
$O(n)$: 1,000 operations
$O(n \log n)$: ~10,000 operations
$O(n^2)$: 1,000,000 operations
$O(2^n)$: 2^1000 operations (practically infinite)
$O(n!)$: 1000! operations (impossible)

Detailed Analysis of Each Class

O(1) - Constant Time

csharp

```csharp
// Array access
public static int GetFirst(int[] arr)
{
    return arr[0]; // Always 1 operation regardless of array size
}

// Hash table operations (average case)
public static void AddToHashSet(HashSet<int> set, int value)
{
    set.Add(value); // O(1) average case
}

// Mathematical calculations
public static int Calculate(int n)
{
    return n * 2 + 5; // Always same number of operations
}
```

O(log n) - Logarithmic Time

csharp

```csharp
// Binary search
public static int BinarySearch(int[] sortedArr, int target)
{
    int left = 0, right = sortedArr.Length - 1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (sortedArr[mid] == target)
            return mid;

        if (sortedArr[mid] < target)
            left = mid + 1;  // Eliminate left half
        else
            right = mid - 1; // Eliminate right half
    }

    return -1;
}
// Each iteration eliminates half the remaining elements
// log₂(n) iterations maximum
```

O(n) - Linear Time

csharp

```csharp
// Single loop through array
public static int Sum(int[] arr)
{
    int sum = 0;
    foreach (int num in arr) // n iterations
    {
        sum += num;
    }
    return sum;
}

// Multiple loops (not nested)
public static void PrintTwice(int[] arr)
{
    foreach (int num in arr)    // n iterations
        Console.WriteLine(num);

    foreach (int num in arr)    // n iterations
        Console.WriteLine(num * 2);

    // Total: 2n = O(n)
}
```

O(n log n) - Linearithmic Time

csharp

```csharp
// Merge Sort
public static void MergeSort(int[] arr, int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        MergeSort(arr, left, mid);      // T(n/2)
        MergeSort(arr, mid + 1, right); // T(n/2)
        Merge(arr, left, mid, right);   // O(n)
    }
}
// Recurrence: T(n) = 2T(n/2) + O(n) = O(n log n)

// Alternative: Sorting then processing
public static void ProcessSorted(int[] arr)
{
    Array.Sort(arr);         // O(n log n)

    foreach (int num in arr)  // O(n)
        Console.WriteLine(num);

    // Total: O(n log n) + O(n) = O(n log n)
}
```

$O(n^2)$ - Quadratic Time

csharp

```csharp
// Nested loops
public static void PrintPairs(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)       // n iterations
    {
        for (int j = 0; j < arr.Length; j++)   // n iterations each
        {
            Console.WriteLine($"({arr[i]}, {arr[j]})");
        }
    }
    // Total: n × n = n²
}

// Bubble sort
public static void BubbleSort(int[] arr)
{
    for (int i = 0; i < arr.Length - 1; i++)     // n iterations
    {
        for (int j = 0; j < arr.Length - i - 1; j++) // (n-i) iterations
        {
            if (arr[j] > arr[j + 1])
            {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    // Total: (n-1) + (n-2) + ... + 1 = n(n-1)/2 = O(n²)
}
```

O(2ⁿ) - Exponential Time

csharp

```
// Naive Fibonacci (without memoization)
public static int Fibonacci(int n)
{
    if (n <= 1) return n;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
    // Each call branches into 2 calls, depth n
    // Total calls ≈ 2^n
}

// Generating all subsets
public static List<List<int>> GenerateSubsets(int[] arr)
{
    List<List<int>> result = new List<List<int>>();
    GenerateSubsetsHelper(arr, 0, new List<int>(), result);
    return result;
}

private static void GenerateSubsetsHelper(int[] arr, int index,
    List<int> current, List<List<int>> result)
{
    if (index == arr.Length)
    {
        result.Add(new List<int>(current));
        return;
    }

    // Include current element
    current.Add(arr[index]);
    GenerateSubsetsHelper(arr, index + 1, current, result);

    // Exclude current element
    current.RemoveAt(current.Count - 1);
    GenerateSubsetsHelper(arr, index + 1, current, result);
}
// For each element, we have 2 choices: include or exclude
// Total subsets: 2^n
```

Analyzing Loops and Iterations

Single Loops

csharp

```csharp
// Linear - O(n)
for (int i = 0; i < n; i++)
{
    // O(1) operation
}

// Still Linear - O(n)
for (int i = 0; i < n; i += 2)
{
    // O(1) operation
}

// Still Linear - O(n)
for (int i = 0; i < n; i += 100)
{
    // O(1) operation
}

// Logarithmic - O(log n)
for (int i = 1; i < n; i *= 2)
{
    // O(1) operation
}
// i takes values: 1, 2, 4, 8, 16, ..., up to n
// Number of iterations = log₂(n)

// Logarithmic - O(log n)
for (int i = n; i > 0; i /= 2)
{
    // O(1) operation
}
```

Nested Loops

csharp

```
// Quadratic - O(n²)
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        // O(1) operation
    }
}

// Still Quadratic - O(n²)
for (int i = 0; i < n; i++)
{
    for (int j = i; j < n; j++)  // Inner loop runs (n-i) times
    {
        // O(1) operation
    }
}
// Total iterations: n + (n-1) + (n-2) + ... + 1 = n(n+1)/2 = O(n²)

// Cubic - O(n³)
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        for (int k = 0; k < n; k++)
        {
            // O(1) operation
        }
    }
}

// O(n log n)
for (int i = 0; i < n; i++)
{
    for (int j = 1; j < n; j *= 2)  // Inner loop is O(log n)
    {
        // O(1) operation
    }
}
// Outer loop: n times, Inner loop: log n times each
// Total: n × log n = O(n log n)
```

Sequential Loops

```csharp
// O(n) - Add the complexities
for (int i = 0; i < n; i++)
{
    // O(1) operation
}

for (int i = 0; i < n; i++)
{
    // O(1) operation
}
// Total: O(n) + O(n) = O(n)

// O(n²) - Take the maximum
for (int i = 0; i < n; i++)
{
    // O(1) operation - O(n) total
}

for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        // O(1) operation - O(n²) total
    }
}
// Total: max(O(n), O(n²)) = O(n²)
```

Recursion Complexity Analysis

Time Complexity of Recursion

Method 1: Recursion Tree

Draw the tree and count total nodes.

```csharp
```

```
// Fibonacci example
public static int Fibonacci(int n)
{
    if (n <= 1) return n;          // Base case: O(1)
    return Fibonacci(n-1) + Fibonacci(n-2); // Two recursive calls
}

/*
Recursion Tree for Fib(5):
                Fib(5)
               /     \
            Fib(4)     Fib(3)
           /    \     /    \
        Fib(3)  Fib(2) Fib(2) Fib(1)
       /   \   /  \   /  \
    Fib(2) Fib(1) Fib(1) Fib(0) Fib(1) Fib(0)
    /  \
  Fib(1) Fib(0)

Height of tree: n
Each node branches into 2: Total nodes ≈ 2^n
Time complexity: O(2^n)
*/
```

Method 2: Recurrence Relations

Express T(n) in terms of T(smaller values).

csharp

```
// Binary search
public static int BinarySearch(int[] arr, int target, int left, int right)
{
    if (left > right) return -1;                // Base case

    int mid = left + (right - left) / 2;
    if (arr[mid] == target) return mid;

    if (arr[mid] < target)
        return BinarySearch(arr, target, mid + 1, right); // T(n/2)
    else
        return BinarySearch(arr, target, left, mid - 1);  // T(n/2)
}

/*
Recurrence relation: T(n) = T(n/2) + O(1)
Solution: T(n) = O(log n)

Proof:
T(n) = T(n/2) + c
T(n/2) = T(n/4) + c
T(n/4) = T(n/8) + c
...
After k steps: T(n/2^k) = T(1) + k×c

When n/2^k = 1, we have k = log₂(n)
Therefore: T(n) = T(1) + log₂(n) × c = O(log n)
*/
```

Space Complexity of Recursion

Space complexity = Maximum depth of recursion × Space per call

csharp

```
// Factorial - Tail recursion
public static int Factorial(int n)
{
    if (n <= 1) return 1;
    return n * Factorial(n - 1);
}
/*
Call stack:
Factorial(5)
  Factorial(4)
    Factorial(3)
      Factorial(2)
        Factorial(1) -> returns 1
      <- returns 2
    <- returns 6
  <- returns 24
<- returns 120

Maximum stack depth: n
Space per call: O(1)
Total space complexity: O(n)
*/

// Binary tree traversal
public static void InorderTraversal(TreeNode root)
{
    if (root == null) return;

    InorderTraversal(root.Left);   // Recurse left
    Console.WriteLine(root.Data);  // Process
    InorderTraversal(root.Right);  // Recurse right
}
/*
Maximum recursion depth = Height of tree
For balanced tree: O(log n) space
For skewed tree: O(n) space
*/
```

Optimizing Recursive Solutions

Memoization (Top-Down DP)

csharp

```
private static Dictionary<int, long> memo = new Dictionary<int, long>();

public static long FibonacciMemo(int n)
{
    if (n <= 1) return n;

    if (memo.ContainsKey(n))
        return memo[n];

    memo[n] = FibonacciMemo(n - 1) + FibonacciMemo(n - 2);
    return memo[n];
}
/*
Each subproblem computed only once
Time complexity: O(n)
Space complexity: O(n) - for memoization + O(n) for recursion = O(n)
*/
```

Data Structure Complexities

Array Operations

csharp

```
int[] arr = new int[1000];

// Access by index: O(1)
int value = arr[5];

// Search (unsorted): O(n)
bool found = Array.Exists(arr, x => x == target);

// Search (sorted): O(log n)
int index = Array.BinarySearch(arr, target);

// Insert at end: O(1) if space available
// Insert at beginning/middle: O(n) - need to shift elements
```

Dynamic Array (List<T>) Operations

csharp

```csharp
List<int> list = new List<int>();

// Add to end: O(1) amortized, O(n) worst case (when resizing)
list.Add(value);

// Insert at index: O(n) - need to shift elements
list.Insert(0, value);

// Remove by index: O(n) - need to shift elements
list.RemoveAt(0);

// Remove by value: O(n) - search + shift
list.Remove(value);

// Access by index: O(1)
int value = list[5];
```

Hash Table (Dictionary<K,V>) Operations

```csharp
Dictionary<string, int> dict = new Dictionary<string, int>();

// Insert: O(1) average, O(n) worst case (hash collisions)
dict["key"] = value;

// Search: O(1) average, O(n) worst case
bool exists = dict.ContainsKey("key");

// Delete: O(1) average, O(n) worst case
dict.Remove("key");

// Access: O(1) average, O(n) worst case
int value = dict["key"];
```

Linked List Operations

```csharp
```

```
// Insert at head: O(1)
void InsertAtHead(Node newNode)
{
    newNode.Next = head;
    head = newNode;
}

// Insert at tail: O(n) for singly linked list, O(1) if tail pointer maintained
void InsertAtTail(Node newNode)
{
    if (head == null) { head = newNode; return; }

    Node current = head;
    while (current.Next != null)  // O(n) traversal
        current = current.Next;

    current.Next = newNode;
}

// Search: O(n)
bool Search(int value)
{
    Node current = head;
    while (current != null)
    {
        if (current.Data == value) return true;
        current = current.Next;
    }
    return false;
}

// Delete: O(n) to find + O(1) to delete = O(n)
```

Stack Operations

csharp

```
Stack<int> stack = new Stack<int>();

// Push: O(1)
stack.Push(value);

// Pop: O(1)
int top = stack.Pop();

// Peek: O(1)
int top = stack.Peek();

// Search: O(n) - not efficiently supported
```

Queue Operations

csharp

```
Queue<int> queue = new Queue<int>();

// Enqueue: O(1)
queue.Enqueue(value);

// Dequeue: O(1)
int front = queue.Dequeue();

// Peek: O(1)
int front = queue.Peek();

// Search: O(n) - not efficiently supported
```

Binary Search Tree Operations

csharp

```
// For balanced BST:
// Search: O(log n)
// Insert: O(log n)
// Delete: O(log n)

// For unbalanced BST (worst case - linear tree):
// Search: O(n)
// Insert: O(n)
// Delete: O(n)

public bool Search(TreeNode root, int value)
{
    if (root == null) return false;

    if (root.Data == value) return true;

    if (value < root.Data)
        return Search(root.Left, value);  // Go left
    else
        return Search(root.Right, value); // Go right
}
```

Algorithm Complexity Examples

Sorting Algorithms Comparison

Algorithm	Best Case	Average Case	Worst Case	Space
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)

Detailed Analysis Examples

Merge Sort Analysis

csharp

```
public static void MergeSort(int[] arr, int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        MergeSort(arr, left, mid);       // T(n/2)
        MergeSort(arr, mid + 1, right);  // T(n/2)
        Merge(arr, left, mid, right);    // O(n)
    }
}

/*
Recurrence relation: T(n) = 2T(n/2) + O(n)

Using Master Theorem:
a = 2, b = 2, f(n) = n
n^logba = n^1 = n
Since f(n) = Θ(n^logba), we have T(n) = Θ(n log n)

Space complexity:
- Recursion depth: O(log n)
- Temporary arrays in merge: O(n)
- Total: O(n)
*/
```

Quick Sort Analysis

csharp

```
public static void QuickSort(int[] arr, int low, int high)
{
    if (low < high)
    {
        int pivotIndex = Partition(arr, low, high);  // O(n)

        QuickSort(arr, low, pivotIndex - 1);       // T(?)
        QuickSort(arr, pivotIndex + 1, high);      // T(?)
    }
}

/*
Best/Average case: Pivot divides array roughly in half
T(n) = 2T(n/2) + O(n) = O(n log n)

Worst case: Pivot is always smallest/largest element
T(n) = T(n-1) + O(n) = O(n²)

Space complexity:
- Best/Average: O(log n) recursion depth
- Worst: O(n) recursion depth
*/
```

Search Algorithms

Linear Search

csharp

```
public static int LinearSearch(int[] arr, int target)
{
    for (int i = 0; i < arr.Length; i++)  // O(n) iterations
    {
        if (arr[i] == target)             // O(1) operation
            return i;
    }
    return -1;
}

/*
Time Complexity:
- Best case: O(1) - target is first element
- Average case: O(n) - target is in middle
- Worst case: O(n) - target is last or not present

Space Complexity: O(1) - only using constant extra space
*/
```

Binary Search

csharp

```
public static int BinarySearch(int[] arr, int target)
{
    int left = 0, right = arr.Length - 1;

    while (left <= right)                // O(log n) iterations
    {
        int mid = left + (right - left) / 2; // O(1) operation

        if (arr[mid] == target) return mid;  // O(1) operation

        if (arr[mid] < target)
            left = mid + 1;              // O(1) operation
        else
            right = mid - 1;             // O(1) operation
    }

    return -1;
}

/*
Time Complexity:
- All cases: O(log n)
- Each iteration eliminates half the search space

Space Complexity: O(1) - only using constant extra space

Requirement: Array must be sorted
*/
```

Best, Average, and Worst Case

Understanding Different Cases

Quick Sort Example

csharp

```
public static void QuickSort(int[] arr, int low, int high)
{
    if (low < high)
    {
        int pivot = Partition(arr, low, high);
        QuickSort(arr, low, pivot - 1);
        QuickSort(arr, pivot + 1, high);
    }
}
```

Best Case: O(n log n)

- Pivot always divides array into two equal halves
- Balanced partitioning at each level

Average Case: O(n log n)

- Random pivot selection leads to reasonably balanced partitions
- Expected behavior for most inputs

Worst Case: O(n^2)

- Pivot is always the smallest or largest element
- Input: Already sorted array [1,2,3,4,5] or reverse sorted [5,4,3,2,1]
- Creates unbalanced partitions (one side has n-1 elements, other has 0)

Hash Table Example

```
csharp

Dictionary<string, int> hashTable = new Dictionary<string, int>();
```

Best/Average Case: O(1)

- Good hash function distributes keys uniformly
- Minimal collisions

Worst Case: O(n)

- All keys hash to same bucket (poor hash function)
- Degenerates to linear search through collision chain

Binary Search Tree Example

csharp

```
public class BST
{
    public TreeNode Search(TreeNode root, int value)
    {
        if (root == null || root.Value == value)
            return root;

        if (value < root.Value)
            return Search(root.Left, value);
        return Search(root.Right, value);
    }
}
```

Best/Average Case: O(log n) - Balanced tree **Worst Case: O(n)** - Skewed tree (essentially a linked list)

Amortized Analysis

What is Amortized Analysis?

Amortized analysis gives the average performance per operation over a sequence of operations. It's useful when occasional expensive operations are balanced by many cheap ones.

Dynamic Array Resizing Example

csharp

```
public class DynamicArray
{
    private int[] array;
    private int size;
    private int capacity;

    public DynamicArray()
    {
        capacity = 1;
        array = new int[capacity];
        size = 0;
    }

    public void Add(int value)
    {
        if (size == capacity)
        {
            // Resize: O(n) operation
            capacity *= 2;
            int[] newArray = new int[capacity];
            for (int i = 0; i < size; i++)
            {
                newArray[i] = array[i];
            }
            array = newArray;
        }

        array[size] = value; // O(1) operation
        size++;
    }
}
```

Analysis:

- Most Add operations: O(1)
- Resize operations: O(n) but infrequent
- Resizing happens at sizes: 1, 2, 4, 8, 16, 32, ...
- For n operations, total resize cost: 1 + 2 + 4 + 8 + ... + n = 2n - 1
- Amortized cost per operation: (n + 2n - 1) / n = O(1)

Stack with Min Operation

```csharp
public class MinStack
{
    private Stack<int> stack;
    private Stack<int> minStack;

    public MinStack()
    {
        stack = new Stack<int>();
        minStack = new Stack<int>();
    }

    public void Push(int value)
    {
        stack.Push(value);
        if (minStack.Count == 0 || value <= minStack.Peek())
            minStack.Push(value);
    }

    public int Pop()
    {
        int value = stack.Pop();
        if (value == minStack.Peek())
            minStack.Pop();
        return value;
    }

    public int GetMin() => minStack.Peek(); // O(1)
}
```

All operations (Push, Pop, GetMin) are O(1) amortized.

Common Mistakes and Pitfalls

Mistake 1: Confusing Different Variables

```csharp
```

```
// WRONG: This is NOT O(n²)
public void PrintArrays(int[] arrayA, int[] arrayB)
{
    foreach (int a in arrayA)       // a iterations
        Console.WriteLine(a);

    foreach (int b in arrayB)       // b iterations
        Console.WriteLine(b);
}
// CORRECT: This is O(a + b) where a and b are different sizes
```

Mistake 2: Ignoring Inner Loop Dependency

csharp

```
// Analyzing this correctly:
for (int i = 0; i < n; i++)
{
    for (int j = i; j < n; j++)  // Inner loop depends on i
    {
        Console.WriteLine($"{i}, {j}");
    }
}

// WRONG thinking: "Inner loop runs n times, so O(n²)"
// CORRECT analysis:
// i=0: inner runs n times
// i=1: inner runs (n-1) times
// i=2: inner runs (n-2) times
// ...
// Total: n + (n-1) + (n-2) + ... + 1 = n(n+1)/2 = O(n²)
```

Mistake 3: Misunderstanding Recursion Space

csharp

```csharp
public int Factorial(int n)
{
    if (n <= 1) return 1;
    return n * Factorial(n - 1);
}

// WRONG: "No extra arrays created, so O(1) space"
// CORRECT: Each recursive call uses stack space
// Maximum n calls on stack simultaneously = O(n) space
```

Mistake 4: Not Considering Best/Worst Cases

csharp

```csharp
// Linear search - don't just say "O(n)"
public static int LinearSearch(int[] arr, int target)
{
    for (int i = 0; i < arr.Length; i++)
    {
        if (arr[i] == target) return i;
    }
    return -1;
}

// COMPLETE analysis:
// Best case: O(1) - found at first position
// Average case: O(n) - found in middle on average
// Worst case: O(n) - not found or at last position
```

Practice Problems

Problem 1: Analyze This Code

csharp

```
public static void Mystery1(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)
    {
        for (int j = 0; j < 1000; j++)
        {
            Console.WriteLine($"{arr[i]}, {j}");
        }
    }
}
```

Answer:

- Time: O(n) - outer loop n times, inner loop constant 1000 times
- Space: O(1) - no extra space used

Problem 2: Analyze This Code

csharp

```
public static void Mystery2(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)
    {
        for (int j = i; j < arr.Length; j++)
        {
            for (int k = 0; k < j; k++)
            {
                Console.WriteLine($"{i}, {j}, {k}");
            }
        }
    }
}
```

Answer:

- Time: $O(n^3)$ - three nested loops, innermost depends on j
- Space: O(1) - no extra space used

Problem 3: Recursive Analysis

csharp

```
public static void PrintBinary(int n)
{
    if (n > 0)
    {
        PrintBinary(n / 2);
        Console.Write(n % 2);
    }
}
```

Answer:

- Time: O(log n) - divides n by 2 each time until it reaches 0
- Space: O(log n) - recursion stack depth is log n

Problem 4: Two-Pointer Technique

csharp

```
public static bool HasPairWithSum(int[] sortedArr, int target)
{
    int left = 0, right = sortedArr.Length - 1;

    while (left < right)
    {
        int sum = sortedArr[left] + sortedArr[right];
        if (sum == target) return true;
        else if (sum < target) left++;
        else right--;
    }
    return false;
}
```

Answer:

- Time: O(n) - each element visited at most once
- Space: O(1) - only using two pointers

Problem 5: Sliding Window

csharp

```
public static int MaxSubarraySum(int[] arr, int k)
{
    int maxSum = 0, windowSum = 0;

    // First window
    for (int i = 0; i < k; i++)
        windowSum += arr[i];

    maxSum = windowSum;

    // Slide the window
    for (int i = k; i < arr.Length; i++)
    {
        windowSum = windowSum - arr[i - k] + arr[i];
        maxSum = Math.Max(maxSum, windowSum);
    }

    return maxSum;
}
```

Answer:

- Time: O(n) - each element added and removed exactly once
- Space: O(1) - only using few variables

Master Method for Divide and Conquer

Master Theorem Formula

For recurrence relations of the form: **T(n) = aT(n/b) + f(n)**

Where:

- **a ≥ 1** (number of subproblems)
- **b > 1** (factor by which subproblem size is reduced)
- **f(n)** (cost of work done outside recursive calls)

Three Cases

Case 1: If f(n) = O(n^(log_b(a) - ε)) for some ε > 0 Then: **T(n) = Θ(n^(log_b(a)))**

Case 2: If f(n) = Θ(n^(log_b(a))) Then: **T(n) = Θ(n^(log_b(a)) log n)**

Case 3: If f(n) = Ω(n^(log_b(a) + ε)) for some ε > 0, and af(n/b) ≤ cf(n) for some c < 1 Then: **T(n) = Θ(f(n))**

Examples

Merge Sort: T(n) = 2T(n/2) + n

- a = 2, b = 2, f(n) = n
- n^(log_b(a)) = n^(log_2(2)) = n^1 = n
- f(n) = n = Θ(n^(log_b(a)))
- **Case 2 applies: T(n) = Θ(n log n)**

Binary Search: T(n) = T(n/2) + 1

- a = 1, b = 2, f(n) = 1
- n^(log_b(a)) = n^(log_2(1)) = n^0 = 1
- f(n) = 1 = Θ(n^(log_b(a)))
- **Case 2 applies: T(n) = Θ(log n)**

Strassen's Matrix Multiplication: T(n) = 7T(n/2) + n^2

- a = 7, b = 2, f(n) = n^2
- n^(log_b(a)) = n^(log_2(7)) ≈ n^2.81
- f(n) = n^2 = O(n^(2.81 - 0.81)) = O(n^(log_b(a) - ε)) where ε = 0.81
- **Case 1 applies: T(n) = Θ(n^(log_2(7))) ≈ Θ(n^2.81)**

Advanced Complexity Concepts

Space-Time Tradeoffs

Often we can reduce time complexity by using more space, or vice versa.

Example: Two Sum Problem

csharp

```csharp
// Approach 1: Brute Force - O(n^2) time, O(1) space
public static int[] TwoSum1(int[] nums, int target)
{
    for (int i = 0; i < nums.Length; i++)
    {
        for (int j = i + 1; j < nums.Length; j++)
        {
            if (nums[i] + nums[j] == target)
                return new int[] { i, j };
        }
    }
    return new int[] { };
}

// Approach 2: Hash Table - O(n) time, O(n) space
public static int[] TwoSum2(int[] nums, int target)
{
    Dictionary<int, int> map = new Dictionary<int, int>();

    for (int i = 0; i < nums.Length; i++)
    {
        int complement = target - nums[i];
        if (map.ContainsKey(complement))
            return new int[] { map[complement], i };
        map[nums[i]] = i;
    }
    return new int[] { };
}
```

Tradeoff Analysis:

- Method 1: Less space but slower for large inputs
- Method 2: More space but much faster for large inputs
- Choice depends on constraints (available memory vs. time requirements)

Cache Complexity

Modern computers have memory hierarchies that affect real performance.

csharp

```
// Cache-friendly: Access elements sequentially
for (int i = 0; i < matrix.GetLength(0); i++)
    for (int j = 0; j < matrix.GetLength(1); j++)
        sum += matrix[i, j];  // Row-major access

// Cache-unfriendly: Access elements by columns
for (int j = 0; j < matrix.GetLength(1); j++)
    for (int i = 0; i < matrix.GetLength(0); i++)
        sum += matrix[i, j];  // Column-major access
```

Both have $O(n^2)$ complexity, but first version runs faster due to better cache locality.

Complexity Analysis Cheat Sheet

Loop Analysis Quick Reference

Loop Pattern	Complexity
`for (i = 0; i < n; i++)`	O(n)
`for (i = 0; i < n; i += 2)`	O(n)
`for (i = 1; i < n; i *= 2)`	O(log n)
`for (i = n; i > 0; i /= 2)`	O(log n)
Nested: `O(f(n))` inside `O(g(n))`	O(f(n) × g(n))
Sequential: `O(f(n))` then `O(g(n))`	O(f(n) + g(n))

Common Recurrence Relations

Recurrence	Solution	Example
T(n) = T(n-1) + O(1)	O(n)	Linear search recursively
T(n) = T(n-1) + O(n)	$O(n^2)$	Selection sort
T(n) = T(n/2) + O(1)	O(log n)	Binary search
T(n) = T(n/2) + O(n)	O(n)	Finding max in array
T(n) = 2T(n/2) + O(1)	O(n)	Tree traversal
T(n) = 2T(n/2) + O(n)	O(n log n)	Merge sort
T(n) = T(n-1) + T(n-2) + O(1)	$O(2^n)$	Fibonacci (naive)

Data Structure Operation Complexities

Operation	Array	Dynamic Array	Linked List	Hash Table	BST (balanced)
Access	O(1)	O(1)	O(n)	N/A	O(log n)
Search	O(n)	O(n)	O(n)	O(1) avg	O(log n)
Insert	N/A	O(1) amortized	O(1) at head	O(1) avg	O(log n)
Delete	N/A	O(n)	O(1) with ref	O(1) avg	O(log n)

Algorithm Complexity Rankings (Best to Worst)

1. **O(1)** - Constant: Hash table access, array indexing
2. **O(log n)** - Logarithmic: Binary search, balanced BST operations
3. **O(n)** - Linear: Simple loops, linear search
4. **O(n log n)** - Linearithmic: Efficient sorting (merge, heap, quick)
5. **O(n^2)** - Quadratic: Simple nested loops, bubble sort
6. **O(n^3)** - Cubic: Triple nested loops, naive matrix multiplication
7. **O(2^n)** - Exponential: Recursive fibonacci, subset generation
8. **O(n!)** - Factorial: Permutation generation, traveling salesman brute force

---

Final Tips for Complexity Analysis

1. Practice Recognition Patterns

Learn to quickly identify common patterns:

- Single loop = likely O(n)
- Nested loops = likely O(n^2)
- Divide by 2 repeatedly = likely O(log n)
- Recursive calls that branch = likely exponential

2. Always Consider Both Time and Space

csharp

```csharp
// Don't forget space analysis
public static int[] CreateSortedCopy(int[] arr)
{
    int[] copy = new int[arr.Length];  // O(n) space
    Array.Copy(arr, copy, arr.Length);
    Array.Sort(copy);                  // O(n log n) time
    return copy;
}
// Time: O(n log n), Space: O(n)
```

3. Watch for Hidden Complexities

csharp

```csharp
// This might look like O(n), but string concatenation is O(n)
string result = "";
foreach (string s in strings)
{
    result += s;  // O(n) operation due to string immutability
}
// Total: O(n²) time complexity

// Better approach:
StringBuilder sb = new StringBuilder();
foreach (string s in strings)
{
    sb.Append(s);  // O(1) amortized
}
string result = sb.ToString();  // O(n)
// Total: O(n) time complexity
```

4. Consider Real-World Constraints

- For small inputs ($n < 100$), $O(n^2)$ might be fine
- For large inputs ($n > 10^6$), need $O(n)$ or $O(n \log n)$
- Memory constraints might favor time-inefficient but space-efficient algorithms

5. Practice Makes Perfect

The key to mastering complexity analysis is practice. Start with simple examples and gradually work your way up to more complex algorithms. Always try to:

1. Identify the input size parameter(s)

2. Count the basic operations
3. Express the count as a function of input size
4. Apply Big O rules to simplify
5. Consider both best and worst cases
6. Don't forget space complexity

Remember: Understanding complexity helps you write better code and ace technical interviews!