Algorithm Analysis and Complexity

Lecture 02

Measuring Efficiency Through Asymptotic and Amortized Analysis

Pranshu Mandal

The Fundamental Question

How do we know if our solution is good?

- Is it fast enough?
- Does it use too much memory?
- Will it scale from 100 to 100,000 items?

We need a scientific way to measure and compare algorithms.

Multiple Solutions, Different Performance

Problem: Find an Element in a List

Solution 1: Linear Search

- Start from first element
- Check each element one by one
- Stop when found or reach end

Solution 2: Binary Search (sorted list)

- Start in the middle
- If target is smaller, search left half
- If target is larger, search right half
- Repeat until found

Question: Which is better? How much better?

Performance Comparison

Searching for 77 in 1,000 sorted numbers:

Linear Search:

```
Check position 1, 2, 3, ..., 547: Found!
Required: 547 comparisons
```

Binary Search:

```
Step 1: Check middle (position 500)
Step 2: Check middle of right half (position 750)
...
Step 10: Found at position 547!
Required: 10 comparisons
```

Result: 547 vs 10 comparisons = **54× faster!**

Types of Analysis

Time Complexity

How does running time grow as input size increases?

Space Complexity

How much memory does the algorithm use?

Best, Average, Worst Case

- **Best case:** Minimum time needed (most optimistic)
- **Average case:** Expected time for typical inputs
- **Worst case:** Maximum time needed (most pessimistic)

Example: Searching Unsorted Array

```
for (int i = 0; i < n; i++) {
    if (arr[i] == target) return i;
}
```

Best case: Element at position 0

→ 1 comparison

Average case: Element somewhere in middle

→ n/2 comparisons

Worst case: Element at end or doesn't exist

→ n comparisons

Machine-Independent Analysis

Problem with measuring actual time:

```
clock_t start = clock();
// ... run algorithm ...
clock_t end = clock();
```

Issues:

- Different computers = different results
- Same computer, different times
- Depends on language, compiler, OS load

Solution: Count Operations

Compare fundamental operations: comparisons, assignments, arithmetic

Asymptotic Analysis

The Big Picture: Growth Rates Matter

How will your algorithm cope as data grows?

Input (n)	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(2^n)$
10	1	3	10	30	100	1,024
100	1	7	100	664	10,000	1.27×10^{30}
1,000	1	10	1,000	9,966	1,000,000	Too big!

Growth rate matters enormously for large inputs!

Big-O Notation

Definition:

Describes the **upper bound** of algorithm's growth rate (worst case)

Practical Meaning:

"The algorithm's running time grows at most as fast as f(n), ignoring constant factors"

Notation: O(f(n))

Common Time Complexities

$O(1)$ - Constant Time

$O(\log n)$ - Logarithmic Time

$O(n)$ - Linear Time

$O(n \log n)$ - Linearithmic Time

$O(n^2)$ - Quadratic Time

$O(2^n)$ - Exponential Time

Let's explore each with examples...

O(1) - Constant Time

Performance doesn't depend on input size

```
int get_element(int arr[], int index) {
    return arr[index];  // One operation
}
```

Whether array has 10 or 10,000,000 elements:

Accessing `arr[5]` takes the same time

Real-world analogy: Opening a specific page in a book by page number

O(log n) - Logarithmic Time

Performance increases slowly as input grows

Doubling input adds only one operation

Example: Binary search

Why O(log n)?

Each step cuts search space in half:

```
1000 → 500 → 250 → 125 → ... → 1
Steps = log₂(1000) ≈ 10
```

Real-world analogy: Finding word in dictionary by halving search space

O(n) - Linear Time

Performance grows proportionally with input

Double input = double time

```
int find_max(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {    // n times
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

Must check every element once → n operations

O(n log n) - Linearithmic Time

Common in efficient sorting algorithms

Examples:

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

Why O(n log n)?

Dividing problem (log n) + processing all elements at each level (n)

Real-world analogy: Organizing deck of cards using divide-and-conquer

$O(n^2)$ - Quadratic Time

Performance proportional to square of input

Very slow for large inputs

```
void print_pairs(int arr[], int n) {
    for (int i = 0; i < n; i++) {          // n times
        for (int j = 0; j < n; j++) {      // n times
            printf("(%d, %d) ", arr[i], arr[j]);
        }
    }
}
// Total: n × n = n²
```

Real-world analogy: Comparing every person with every other person in a room of 100 = 10,000 comparisons!

$O(2^n)$ - Exponential Time

Performance doubles with each additional element

Impractical for even moderate inputs

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}
// Creates exponentially growing tree of calls
```

For n=30: over **1 billion** function calls!

Real-world analogy: Chain letter where each person sends to 2 others

Growth Rate Visualization

```
Operations
    |
10⁶ |                                      • O(2ⁿ)
    |
10⁵ |                             • O(n²)
    |
10⁴ |                    • O(n log n)
    |
10³ |          • O(n)
    |
10² |    • O(log n)
    | • O(1)
    |_________________
      10  100  1000  ... Input Size (n)
```

Simplification Rules for Big-O

Rule 1: Drop Constants

$O(2n) \rightarrow O(n)$

$O(500) \rightarrow O(1)$

$O(n/2 + 100) \rightarrow O(n)$

Why?

Constants don't affect growth rate

For large n, whether it's 2n or 5n doesn't matter vs n^2

Rule 2: Drop Lower-Order Terms

$O(n^2 + n) \rightarrow O(n^2)$

$O(n^2 + 100n + 500) \rightarrow O(n^2)$

$O(n \log n + n) \rightarrow O(n \log n)$

Why?

Highest-order term dominates for large n

Example: n = 1,000,000

- n^2 = 1,000,000,000,000
- n = 1,000,000
- The n term is negligible!

Rule 3: Different Variables Stay Different

$O(m + n)$ or **$O(m \times n)$** for two different inputs

Don't simplify different variables!

If you know $m = n$, then $O(n^2)$

If m is constant, then $O(n)$

But without knowing, keep both variables

Analyzing Code Examples

Example 1: Single Loop

```
for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);  // O(1) operation
}
```

Loop runs n times → **O(n)**

Example 2: Nested Loops

```
for (int i = 0; i < n; i++) {                // n times
    for (int j = 0; j < n; j++) {            // n times
        printf("%d ", arr[i] + arr[j]); // O(1)
    }
}
```

Outer: n times

Inner: n times for each outer

Total: $n \times n \rightarrow O(n^2)$

Example 3: Sequential Statements

```c
// Part 1
for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
}

// Part 2
for (int i = 0; i < n; i++) {
    printf("%d ", arr[i] * 2);
}
```

Part 1: $O(n)$

Part 2: $O(n)$

Total: $O(n) + O(n) = O(2n) \rightarrow$ **$O(n)$** (drop constant)

Example 4: Logarithmic Pattern

```c
int i = 1;
while (i < n) {
    printf("%d ", i);
    i = i * 2;  // i doubles each iteration
}
```

Values of i: 1, 2, 4, 8, 16, ..., n

Number of iterations: $\log_2(n)$ → **O(log n)**

Example 5: Different Inputs

```c
for (int i = 0; i < n; i++) {       // n times
    for (int j = 0; j < m; j++) {  // m times
        printf("%d ", i + j);
    }
}
```

Two different input sizes → **O(n × m)**

Don't simplify without knowing relationship!

Other Important Notations

Big-Omega (Ω): Lower Bound

Ω(f(n)): Algorithm takes **at least** f(n) time

Example: Finding element in unsorted array

- **O(n):** Worst case - check all n elements
- **Ω(1):** Best case - element at position 0

Big-Theta (Θ): Tight Bound

Θ(f(n)): Algorithm takes **exactly** f(n) time

(Both upper and lower bounds)

Example: Printing all array elements

```
for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
}
```

- Must visit every element: Ω(n)
- Visits each exactly once: O(n)
- Therefore: **Θ(n)**

Relationship Between Notations

Big-O: Upper bound (≤)

Big-Omega: Lower bound (≥)

Big-Theta: Tight bound (=)

Analogy:

- **$O(n^2)$:** "Commute takes at most 1 hour"
- **$\Omega(n^2)$:** "Commute takes at least 30 minutes"
- **$\Theta(n^2)$:** "Commute takes exactly 45 minutes"

Amortized Analysis

Why Amortized Analysis?

Problem: Some operations have **varying costs**

Example: Parking lot

- 9 out of 10 times: instant parking
- Every 10th car: reorganize lot (expensive!)

Question: What's the "average" cost per car?

Can't just say "worst case = O(reorganization)" because most cars don't pay that cost!

What is Amortized Analysis?

Computes average cost per operation over a sequence

Key Idea: Spread cost of expensive operations over many cheap ones

Important Distinction:

- **Average-case:** Probability distribution of inputs
- **Amortized:** Sequence of operations, guarantees average

Classic Example: Dynamic Array Resizing

Problem:

Arrays are fixed-size. What when full and need to add more?

Solution:

When full:

1. Allocate new array of **double** the size
2. Copy all elements to new array
3. Free old array
4. Add new element

Question: What's the cost of insertion?

Step-by-Step Trace

Starting with capacity 1, insert elements 1-8:

Insert #	Capacity	Resize?	Copy Cost	Insert Cost	Total
1	1	No	0	1	1
2	1	Yes	1	1	2
3	2	Yes	2	1	3
4	4	No	0	1	1
5	4	Yes	4	1	5
6	8	No	0	1	1
7	8	No	0	1	1
8	8	No	0	1	1

Cumulative total for 8 insertions: 15 operations

Individual vs Amortized Cost

Individual operation costs:

- Best case: 1 (just insert, no resize)
- Worst case: n (resize requires copying n elements)

Naive conclusion: "Insertion is O(n), dynamic arrays are slow!"

But look at the sequence:

- 8 insertions → 15 total operations
- Average: $15/8 \approx 1.875$ per insertion
- This is approximately **constant time**!

The Mathematical Analysis

For n insertions with doubling:

Total cost = n (insertions) + (1 + 2 + 4 + 8 + … + n/2) (copies)

Geometric series sum: $1 + 2 + 4 + \ldots + n/2 < 2n$

Total cost $< n + 2n = 3n$

Average per insertion = $3n / n = 3$

Amortized time: $O(1)$

Cost Distribution Visualization

```
Individual Insertion Costs:
Insert: 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
Cost:   1  2  3  1  5  1  1  1  9  1   1   1   1   1   1   1
           ^     ^        ^                                 ^
        Resize Resize  Resize                          Resize

Most operations: Cheap (cost = 1)
Occasional operations: Expensive (cost = n)
Average over all: Constant (≈ 3)
```

Why This Matters

Without amortized analysis:

"Dynamic arrays have O(n) insertion. Use linked lists!"

With amortized analysis:

"Dynamic arrays have O(1) amortized insertion. Very efficient!"

Real-world usage:

- Python's `list`
- Java's `ArrayList`
- C++'s `std::vector`

All use dynamic arrays based on amortized analysis!

The Banking Method

Think of it like a bank account:

Cheap operations: Deposit extra credits
Expensive operations: Withdraw from saved credits

For dynamic arrays:

- Each insertion "costs" 3 credits (amortized)
- Cheap insertion: Use 1, save 2
- Expensive insertion: Use 1 + saved credits for copying

Saved credits from cheap operations pay for expensive ones!

Real Implementation

```c
typedef struct {
    int *arr;
    int size;       // Current elements
    int capacity;   // Current capacity
} DynamicArray;

void insert(DynamicArray *da, int value) {
    if (da->size == da->capacity) {
        // Resize (expensive!)
        int new_cap = da->capacity * 2;
        int *new_arr = malloc(new_cap * sizeof(int));
        for (int i = 0; i < da->size; i++) {
            new_arr[i] = da->arr[i];
        }
        free(da->arr);
        da->arr = new_arr;
        da->capacity = new_cap;
    }
    da->arr[da->size++] = value;  // Always cheap
}
```

Practice Problems

Problem 1

```
int sum = 0;
for (int i = 0; i < n; i++) {
    sum += arr[i];
}
```

Answer: One loop, n iterations → **O(n)**

Problem 2

```
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        printf("%d ", arr[i] + arr[j]);
    }
}
```

Analysis:

- i=0: inner runs n times
- i=1: inner runs n-1 times
- Total: n + (n-1) + (n-2) + ... + 1 = n(n+1)/2

Answer: O(n^2)

Problem 3

```c
int i = n;
while (i > 1) {
    printf("%d ", i);
    i = i / 2;
}
```

Analysis: i starts at n, halves until 1

Answer: O(log n)

Comparison of Data Structures

Data Structure	Access	Search	Insert	Delete
Array	O(1)	O(n)	O(n)	O(n)
Sorted Array	O(1)	O(log n)	O(n)	O(n)
Linked List	O(n)	O(n)	O(1)*	O(1)*
Hash Table	N/A	O(1)**	O(1)**	O(1)**
BST	O(log n)***	O(log n)***	O(log n)***	O(log n)***

*With pointer to position

**Average case

***Balanced trees

Key Takeaways

What We Learned

Why Analyze Algorithms?

- Multiple solutions exist
- Need objective comparison
- Performance matters at scale

Asymptotic Analysis

- Big-O: upper bound (worst case)
- Big-Omega: lower bound (best case)
- Big-Theta: tight bound (exact)
- Common: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$

What We Learned (continued)

Simplification Rules

- Drop constant factors
- Drop lower-order terms
- Keep different variables separate

Amortized Analysis

- Average cost over sequence
- Different from average-case
- Example: Dynamic array O(1) amortized