Asymptotic Analysis

Why Algorithm Efficiency Matters

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

How Do We Measure "Fast"?

- You write two different algorithms that solve the same problem
- **Which one is better?**

We could just run them both and time them...

- But what input size should we test?
- What computer should we use?
- How do we compare results across different machines?

We need a better way to compare algorithms

Our Experiment Setup

Three Algorithms, One Task

1. Linear Search

Find a specific number in an array

$O(n)$

2. Standard Sort

Efficient sorting (merge sort)

$O(n \log n)$

3. Bubble Sort

Simple but slow sorting

$O(n^2)$

Goal: Measure how long each takes as we increase the input size

Algorithm 1: Bubble Sort

How It Works

The idea:

- Compare adjacent elements
- Swap if wrong order
- Repeat until sorted

Example: `[7, 3, 5, 2]`

- Pass 1:
  - `[7,3,5,2]` → `[3,7,5,2]`
  - `[3,7,5,2]` → `[3,5,7,2]`
  - `[3,5,7,2]` → `[3,5,2,7]`
- Pass 2: `[3,5,2,7]` → `[3,2,5,7]`
- Pass 3: `[2,3,5,7]` ✓

```
void bubbleSort(vector<int>& arr) {
  int n = arr.size();
  for (int i = 0; i < n-1; i++) {
    for (int j = 0; j < n-i-1; j++) {
      if (arr[j] > arr[j+1]) {
        swap(arr[j], arr[j+1]);
      }
    }
  }
}
```

Two nested loops → $n \times n = n^2$

Algorithm 2: Linear Search

How It Works

The idea:

- Look at each element
- Stop when found
- Or reach the end

Find 5 in `[7, 3, 5, 2, 8]`

- Check 7? No.
- Check 3? No.
- Check 5? **Yes!**

```
bool linearSearch(
    const vector<int>& arr,
    int target) {
  for (int x : arr) {
    if (x == target) {
      return true;
    }
  }
  return false;
}
```

One loop → n comparisons

Algorithm 3: Standard Sort

How It Works

The idea:

- C++ uses optimized sorting (introsort)
- Hybrid of quicksort, heapsort, insertion sort
- Much smarter than bubble sort

```
void sortArray(vector<int>& arr) {
  sort(arr.begin(), arr.end());
}
```

We don't need all the details

This is what professionals use

Complexity: O(n log n)

How We Measure Time in C++

High Precision Timing

C++ provides high-resolution timers:

```
#include <chrono>

auto start = high_resolution_clock::now();

// ... code we want to time ...

auto end = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(end - start);

cout << "Time: " << duration.count() << " ms" << endl;
```

This gives us millisecond precision

The Experimental Setup

What We'll Measure

For each algorithm:

1. Generate random array of size n
2. Time the algorithm
3. Record the time
4. Repeat for different n values

Input sizes tested:

- **Small:** 1,000, 2,000, 3,000, 5,000
- **Medium:** 10,000, 15,000
- **Large:** 50,000, 100,000, 150,000

What do you predict will happen?

Running the Experiment

Let's See What Happens

The commands:

```
g++ -O2 -std=c++17 compare_algorithms.cpp -o compare_algorithms
./compare_algorithms
```

What it does:

- Tests each algorithm
- Prints timing data
- Generates plots
- Analyzes growth rates

Output preview:

Size	Linear	Sort	Bubble
1000	0	0	0
2000	0	0	1
3000	0	0	4
5000	0	0	10

The Results - Raw Data

What We Measured

Bubble Sort:

n	Time (ms)
1,000	0
2,000	1
3,000	4
5,000	10
7,000	16
10,000	28
15,000	59

Standard Sort:

n	Time (ms)
100,000	6

Notice:

Input 1.5× larger (2000→3000)

Time becomes 4× longer!

The Pattern Emerges

Analyzing Growth Rate

When we change bubble sort input size:

n increases by	Time increases by	Expected for $O(n^2)$
$1.5\times$ (2000→3000)	$4\times$	$2.25\times$
$1.67\times$ (3000→5000)	$2.5\times$	$2.78\times$
$1.5\times$ (10000→15000)	$2.1\times$	$2.25\times$

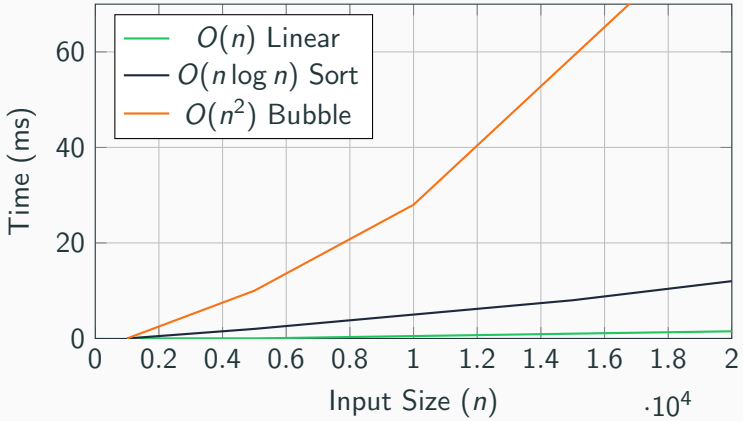
Key Observation

Time increases by roughly the square of size increase

This is the signature of $O(n^2)$ behavior

Visualizing the Difference

Three Growth Rates Compared



The gap between algorithms grows dramatically as n increases

The Dramatic Divergence

Extrapolating to Larger Inputs

At $n = 100,000$:

Algorithm	Time
Linear Search $O(n)$	0 ms (too fast!)
Standard Sort $O(n \log n)$	6 ms
Bubble Sort $O(n^2)$	$\sim$5,900 ms (6 seconds!)

Why such a huge difference?

- Bubble Sort: $100,000^2 =$ **10 billion operations**
- Standard Sort: $100,000 \times \log(100,000) \approx$ **1.7 million operations**
- That's a **6,000× difference!**

The Key Insight

Running Time Depends on Input Size

The same algorithm takes different amounts of time
for different input sizes.

The relationship follows a pattern: $T(n)$

- Bubble Sort: $T(n) \approx c \times n^2$
- Standard Sort: $T(n) \approx c \times n \log n$
- Linear Search: $T(n) \approx c \times n$

The shape of this function matters more than the constant c

Why Constants Don't Matter

Growth Rate vs Constant Factors

Imagine two algorithms:

- Algorithm A: Takes $100n$ operations
- Algorithm B: Takes n^2 operations

For small $n = 10$:

A: $100 \times 10 = 1,000$ ops

B: $10^2 = 100$ ops

B is 10× faster!

For large $n = 10,000$:

A: $100 \times 10,000 = 1$M ops

B: $10,000^2 = 100$M ops

A is 100× faster!

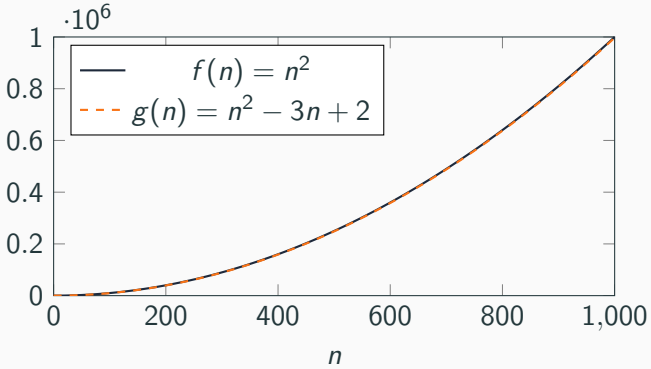
The constant factor (100) becomes irrelevant for large inputs

When Functions Look Similar

A More Subtle Example

Consider these two functions:

- $f(n) = n^2$
- $g(n) = n^2 - 3n + 2$



The Asymptotic View

What "Asymptotic" Means

Asymptotic: Behavior as n approaches infinity

For large enough n:

- $n^2 - 3n + 2 \approx n^2$ (lower terms don't matter)
- $5n^2 + 100n + 50 \approx n^2$ (constants don't matter)
- Both are $O(n^2)$

We care about the rate of growth,
not the exact formula

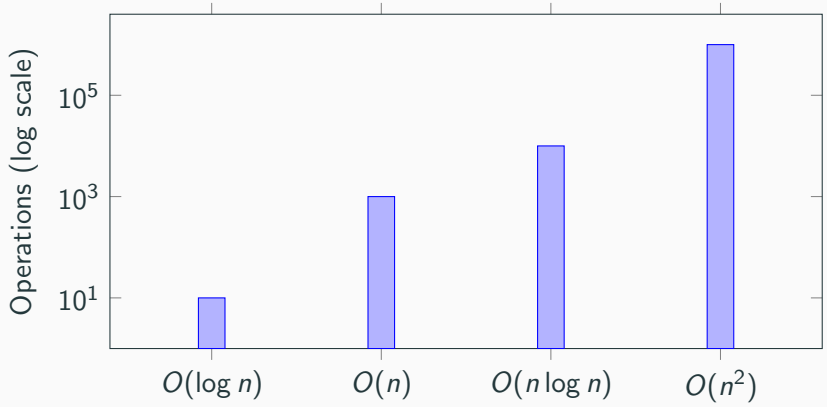
Common Growth Rates

The Complexity Hierarchy

Complexity	Name	Example
$O(1)$	Constant	Array access: `arr[5]`
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Linear search, find max
$O(n \log n)$	Linearithmic	Merge sort, quicksort
$O(n^2)$	Quadratic	Bubble sort, nested loops
$O(2^n)$	Exponential	All subsets (impractical)

How Growth Rates Compare

At $n = 1,000$



Note: $O(2^n)$ would be off the chart - more than atoms in the *observable universe*!

Big-O Notation: Informal Definition

What Does $O(n^2)$ Mean?

$f(n) = O(g(n))$ means "f grows no faster than g"

What we ignore:

- Constant factors: $5n^2 = O(n^2)$
- Lower-order terms: $n^2 + 5n + 3 = O(n^2)$

What we keep:

- The dominant term: n^2

Examples:

- $3n + 5 = O(n)$
- $n^2 + 100n = O(n^2)$
- $2n \log n + n = O(n \log n)$
- $7 = O(1)$

Why This Notation Is Useful

Machine-Independent Comparison

Remember our original question?

Without Big-O:

"Algorithm A takes $5.3n^2 + 2.1n + 7$ ms on my laptop"

Hard to compare, depends on computer

✓ With Big-O:

"Algorithm A is $O(n^2)$"
"Algorithm B is $O(n \log n)$"

Clear: B is better for large n

Big-O lets us reason about algorithms independent of hardware

Back to Our Experiments

Understanding What We Saw

Bubble Sort: $O(n^2)$

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

Total: $n \times n = n^2$ operations

Linear Search: $O(n)$

```
for (i = 0; i < n; i++) {
  // constant work
}
```

Total: n operations

This is why the timings matched our predictions!

Practical Impact

Real-World Consequences

Scenario: Social media feed sorting

Starting: 1,000 users

$O(n^2)$: $\sim$1 ms ✓

$O(n \log n)$: $\sim$0.5 ms ✓

Both look acceptable

Scaling: 1,000,000 users

$O(n^2)$: $\sim$1,000,000 ms

$=$ **16 minutes ✗**

$O(n \log n)$: $\sim$6,000 ms

$=$ **6 seconds ✓**

The algorithmic choice determines if your app is usable at scale

Another Example: Matrices (Why Complexity Matters)

Naive matrix multiplication ($n \times n$ matrices):

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Three nested loops $\rightarrow O(n^3)$

Matrix Size	Approximate Time
100×100	$\sim$1 ms
$1,000 \times 1,000$	$\sim$1 second
$10,000 \times 10,000$	$\sim$16 minutes
$100,000 \times 100,000$	$\sim$11 days!

Log–Log Plots and Power Laws

Why straight lines matter

- Many algorithms follow a **power law**:

$$T(n) \approx c \cdot n^k$$

- Taking logs:

$$\log T(n) = \log c + k \log n$$

- On a **log–log plot**:
  - Power laws become **straight lines**
  - The **slope** of the line is the exponent k

Interpretation:

- Flat line $\Rightarrow$ small growth
- Steep line $\Rightarrow$ fast growth
- Bigger slope $\Rightarrow$ worse asymptotic behavior

Examples on a Log–Log Plot

Algorithm	Slope on log–log plot
Linear search	≈ 1
Bubble sort	≈ 2
Matrix multiplication	≈ 3

Key idea:

- Log–log plots let us **read off Big-O visually**
- We care about the *slope*, not the exact timing

What We've Learned

Key Takeaways

1. **Running time depends on input size:** $T(n)$
2. **Growth rate matters more than constants**
   For large n, $O(n \log n)$ beats $O(n^2)$ regardless
3. **We can measure and verify empirically**
   Experiments matched theoretical predictions
4. **Big-O notation describes growth rates**
   Machine-independent comparison
5. **This has real practical impact**
   Right algorithm = usable application

Try It Yourself

Run the experiments:

```
git clone github.com/amoretti86/data_structures_algorithms
cd data_structures_algorithms/running_times
make run
```

Observe:

- How do measured times grow?
- Do ratios match theoretical predictions?
- What happens with larger inputs?

Experiment:

- Modify code to test other algorithms
- Change input sizes
- Try best-case vs worst-case inputs

What's Next

Building on This Foundation

Today we covered:

- Why we measure $T(n)$
- Common growth rates
- Big-O notation basics
- Empirical verification

Coming up:

- Analyzing loops and recursive functions
- Best case vs worst case vs average case
- Formal proofs and recurrence relations
- Space complexity (memory usage)
- Advanced topics (amortized analysis)