Analyzing Divide and Conquer Algorithms: The Master Method

CS 351: Analysis of Algorithms

Lucas P. Cordova, Ph.D.

This lecture connects abstract algorithms to real applications you might encounter daily. Matrix multiplication powers Netflix recommendations, Google search, video game graphics, and AI systems. We'll discuss how clever algorithm design can dramatically improve performance.

Table of contents

0.1 Executive Summary

What We'll Discuss Today

- How tech companies like Netflix make recommendations so fast
- Why video games can render realistic 3D graphics in real-time
- How a simple change in thinking can make algorithms dramatically faster
- A simple tool for predicting how fast any recursive algorithm will be
- Why some "obvious" solutions aren't actually the best

We'll start with a real problem - how Netflix can recommend movies to millions of users quickly. This leads us to understand how computers multiply large grids of numbers (matrices). We'll see the obvious solution, then learn a powerful tool for analyzing "divide and conquer" solutions, and finally discover a breakthrough that changed computer science.

1 Motivation & Context

1.1 Matrix Multiplication Example

- **Simple Example:** Netflix predicting Alice's rating for "Titanic"
- **Alice's ratings:**

```
Avengers: 5 stars
Romantic Comedy: 2 stars
```

- **Movie features:**

```
Titanic has:
- Action level: 1
- Romance level: 5
```

- **Prediction formula:** Alice's predicted rating = $(5 \times 1) + (2 \times 5) = 15$
- **In general:** Each cell in the result = Row from first grid $\times$ Column from second grid
- **The Computer's Method:**

```python
# For each user (row)
for user in users:
    # For each movie (column)
    for movie in movies:
        rating = 0
        # Combine all features
        for feature in features:
            rating += user[feature] * movie[feature]
```

- **Checkpoint Question 1:** If we have 1000 users, 1000 movies, and 100 features, how many steps does this take?

This breaks down matrix multiplication into an intuitive example using Netflix recommendations. Instead of abstract mathematical notation, we're combining user preferences with movie features produces predictions. The code shows the basic three-loop structure that leads to cubic time complexity.

1.2 Another Example: How Netflix Works

- **The Challenge:** Netflix has 200 million users and thousands of movies.
  - How do they instantly recommend movies you might like?
- **The Secret:**
  - They use **matrix multiplication** - multiplying huge grids of numbers
- **User Ratings Grid:**

	Avengers	Titanic	Comedy1
Alice	5	2	4
Bob	1	5	3
Carol	4	1	5

- **Movie Features Grid:**

```
           Action  Romance
Avengers      5       1
Titanic       1       5
Comedy1       2       3
```

- **The Math:**
  - Multiply these grids to predict what Alice might rate movies she hasn't seen
- **Other Applications:**
  - **Video Games:** 3D graphics and character animation
  - **Google Search:** Ranking web pages
  - **AI/ChatGPT:** Processing language and images
  - **Photo Filters:** Instagram and Snapchat effects
- **The Problem:**
  - With millions of users and movies, these grids become HUGE. How do we multiply them fast enough?

This introduces matrix multiplication through Netflix's recommendation system. The concept is that computers multiply large grids of numbers to solve real problems. We see this in video games for 3D graphics, search engines for ranking, and AI systems for processing data. The challenge is doing this fast enough for real-time applications.

1.3 Why This Matters: The Performance Problem

- **Reality Check:** How long does the basic method take?

Grid Size	Real-World Example	Time with Basic Method
100×100	Small mobile game	0.01 seconds
1,000×1,000	Netflix for small city	17 minutes
10,000×10,000	Netflix nationwide	11.6 DAYS

- **The Question:** Can we do better than the basic approach?
- **Big Idea:** "Divide and Conquer" - Break big problems into smaller pieces
- **Real-World Analogy:** Like organizing a huge library:
  1. Don't sort all books at once
  2. Split into sections (fiction, non-fiction, etc.)

3. Sort each section separately

4. Combine the results

- **For Matrix Math:**

1. Split big grids into smaller blocks
2. Multiply the blocks (recursively)
3. Combine the block results
4. Hope this is faster!

This section shows why performance matters with concrete timing examples that demonstrate the cubic scaling problem. The divide-and-conquer idea is introduced through a library organization analogy that's more accessible than abstract algorithmic concepts.

2 First Attempts

2.1 The Divide-and-Conquer Attempt

The Idea: Treat blocks of numbers like individual numbers

- **Example:** Split a 4×4 grid into four 2×2 blocks

```
Original grids:        Split into blocks:
[a b c d]             [A B]    where A = [a b]
[e f g h]       →     [C D]          B = [c d] etc.
[i j k l]
[m n o p]
```

- **Block Multiplication Rule:** Same pattern as regular multiplication!

```
[A B] × [E F] = [A×E + B×G  A×F + B×H]
[C D]   [G H]   [C×E + D×G  C×F + D×H]
```

- **The Algorithm:**

1. If grid is small (like 1×1), just multiply normally
2. Otherwise, split both grids into 4 blocks each
3. Multiply the 8 needed block pairs (recursive calls)
4. Add the results together

Problem: This needs 8 "recursive calls" - is that actually better?

The divide-and-conquer approach is explained through visual block decomposition rather than mathematical formulas. The key insight is that blocks behave like individual numbers. However, this creates 8 recursive subproblems, and we need to analyze whether this actually helps performance.

2.2 Analyzing Our "Divide and Conquer" Idea

- **The Question:** Is our block method actually faster?
- **Let's Think About It:**
  - We split the problem in half (both dimensions)
  - This creates 8 smaller problems to solve
  - Each smaller problem is 1/4 the size of the original
  - We also need to add the results together (easy part)
- **The Pattern:**

  ```
  Work to solve size n = 8 × (work to solve size n/2) + easy combining work
  ```

- **In Math Notation:**
  - $T(n) = 8T(\frac{n}{2}) + O(n^2)$
- **Need a Tool:** How do we solve this kind of equation to find out if we're actually faster?

> **i** Hypothetical Scenario
>
> What if we had a systematic tool to analyze "divide and conquer" algorithms and told us exactly how fast (or slow) our approach is?

3 Introducing The Tool

3.1 Tool: The Algorithm Analyzer (Master Method)

The Tool: For any "divide and conquer" algorithm, answer these questions:

1. **How many smaller problems do you create?** (call this **a**)
2. **How much smaller is each problem?** (call this **b**)
3. **How much extra work do you do?** (call this **d**)

The Magic Formula: Based on your answers, the tool predicts your algorithm's speed!

> **!** The Three Possible Outcomes
>
> - **Case 1 (Balanced):** Work is spread evenly across all levels
>   - $Time = Size^d \times \log(\text{Size})$
> - **Case 2 (Top-Heavy):** Most work happens at the beginning
>   - $Time = Size^d$
> - **Case 3 (Bottom-Heavy):** Most work happens in the tiny pieces
>   - $Time = Size^{\log_b a}$

Which case applies? Compare **a** with b^d

- If $a = b^d \rightarrow$ Case 1 *(balanced)*
- If $a < b^d \rightarrow$ Case 2 *(top-heavy)*
- If $a > b^d \rightarrow$ Case 3 *(bottom-heavy)*

The comparison between a and b^d determines which case applies, leading to specific formulas for performance.

3.2 Testing Our Algorithm Analyzer

4 Example: Organizing Your Photos

Problem: Sort 1000 photos by date

Divide-and-Conquer Approach:

- Split photos in half $\rightarrow$ 2 smaller problems
- Each problem is half the size $\rightarrow$ divide by 2
- Merging sorted halves takes time proportional to total photos $\rightarrow$ d = 1

Recipe: $a = 2,\ b = 2,\ d = 1$
Analysis: $a = 2,\ b^d = 2^1 = 2$, so $a = b^d \rightarrow$ Case 1
Result: Time $= n \times \log(n)$ — much better than n^2 naive sorting!

---

5 Example: Google Search

Problem: Find "pizza" in a sorted list of 1 billion web pages

Binary Search Approach:

- Look at middle page, go left or right → 1 smaller problem
- Each problem is half the size → divide by 2
- Comparing one word is instant → d = 0

Recipe: $a = 1,\ b = 2,\ d = 0$
Analysis: $a = 1,\ b^d = 2^0 = 1$, so $a = b^d \rightarrow$ Case 1
Result: Time = $\log(n)$ — find anything in ~ 30 steps!

---

6 Example: Our Matrix Block Method

Problem: Netflix recommendation calculation

Our Block Approach:

- Split into 4 blocks, need 8 calculations → a = 8
- Each block is 1/4 the area, so 1/2 the dimension → b = 2
- Combining blocks takes time proportional to n^2 → d = 2

Recipe: $a = 8,\ b = 2,\ d = 2$
Analysis: $a = 8,\ b^d = 2^2 = 4$, so $a > b^d \rightarrow$ Case 3
Result: Time = n^3 — same as the basic method!?

These examples connect the Master Method to familiar real-world problems like organizing photos and Google search. The matrix multiplication example shows why the straightforward divide-and-conquer approach doesn't help - too many recursive calls eliminate any benefit. This sets up the need for a more clever approach.

6.1 Say what? We Didn't Improve Matrix Multiplication?

The Problem: Our block method didn't help because we needed 8 calculations (a = 8)

The Big Question: What if we could do it with only 7 calculations instead of 8?

Seems Impossible!

To multiply blocks:

```
[A B] × [E F] = [? ?]
[C D]   [G H]   [? ?]
```

We need: AE, BG, AF, BH, CE, DG, CF, DH

That's 8 different values - how can 7 calculations possibly be enough?

The Genius Insight (1969):

A mathematician named Strassen found a way!

The Secret: Don't calculate those 8 values directly. Instead:

1. Calculate 7 different, cleverly chosen values
2. Use math tricks to combine them
3. Get the same final answer!

Why This Matters:

- 8 calculations → Recipe gives us a = 8, result = n^3
- 7 calculations → Recipe gives us a = 7, result = n^2.807

That one less calculation breaks the n^3 barrier for the first time in history! This is Strassen's breakthrough.

This section builds drama around Strassen's discovery, emphasizing how reducing just one recursive call has a revolutionary impact. The breakthrough is presented as seeming impossible at first - how can 7 calculations replace 8? This sets up the "magic" of Strassen's algebraic insight.

7 The Breakthrough

7.1 Strassen's Magic Trick Revealed

The Challenge: We need these 8 block multiplications:

AE, BG, AF, BH, CE, DG, CF, DH

Strassen's Secret Recipe: Instead, calculate these 7 special combinations:

Step	What to Calculate	Why This Helps
1	A × (F - H)	Captures A×F relationship
2	(A + B) × H	Captures both A×H and B×H
3	(C + D) × E	Captures both C×E and D×E
4	D × (G - E)	Captures D×G relationship
5	(A + D) × (E + H)	Captures diagonal elements
6	(B - D) × (G + H)	Balances the equations
7	(A - C) × (E + F)	Completes the system

The Reconstruction Magic:

These 7 values contain enough information to figure out all 8 original values through addition and subtraction!

Performance Impact:

- **Old way:** a = 8 → Algorithm Analyzer says n^3 time
- **Strassen's way:** a = 7 → Algorithm Analyzer says $n^{2.807}$ time

Real-World Impact: Netflix calculations that took 1 hour now take 25 minutes!

This presents Strassen's seven products in a more accessible table format, explaining the intuition behind each calculation rather than just listing formulas. The focus is on the breakthrough impact rather than the detailed algebra. The real-world timing comparison makes the improvement tangible.

7.2 Proving Strassen's Breakthrough

Applying Our Algorithm Analyzer to Strassen:

Strassen's Recipe: $T(n) = 7T(\frac{n}{2}) + (\text{combining work})$

The Numbers:

- a = 7 (only 7 recursive calculations!)

- b = 2 (problems are half the size)
- d = 2 (combining takes n^2 steps)

Algorithm Analyzer Says: $a = 7 \quad \text{vs} \quad b^d = 4$

Since $7 > 4$, we have **Case 3 (Bottom-Heavy)**

Final Result: Time = $n^{\log_2 7} = n^{2.807}$

> ❗ Historic Achievement
>
> **First algorithm EVER to beat the "obvious" n^3 approach for matrix multiplication!**

The Impact in Real Numbers:

Problem Size	Old Method	Strassen's Method	Improvement
1,000×1,000 Netflix	17 minutes	4.2 minutes	4× faster
10,000×10,000 Enterprise	11.6 days	1.9 days	6× faster

Why One Less Calculation Matters So Much: The savings compound at every level of recursion!

Checkpoint Question 3: Why does reducing from 8 to 7 calculations have such a dramatic effect?

The table demonstrates real-world impact rather than just abstract complexity. The emphasis is on why this was such a breakthrough - the first algorithm to beat the obvious approach for a fundamental problem!

7.3 Our Algorithm Analyzer in Action

8 Credit Card Processing

Problem: Multiply two very large numbers (like in cryptography)

Smart Algorithm (Karatsuba): Instead of the grade-school method
- Split big numbers in half → **3 calculations** instead of **4**
- Each calculation is on half-sized numbers → $b = 2$
- Combining results takes linear time → $d = 1$

Recipe: $a = 3,\ b = 2,\ d = 1 \rightarrow 3 > 2^1 \rightarrow$ Case 3

Result: $n^{1.59}$ instead of n^2 — saves millions in processing costs!

9 Social Media Analytics

Problem: Process all friendships in Facebook's social graph

Tree Processing Algorithm:
- Visit each person exactly once → **2 recursive calls**
- Each call processes half the network → $b = 2$
- Minimal work per person → $d = 0$

Recipe: $a = 2,\ b = 2,\ d = 0 \rightarrow 2 > 1 \rightarrow$ Case 3

Result: Exactly n steps — optimal since you must visit everyone!

---

10 Smart Search Features

Problem: Find a specific post in your Instagram feed (sorted by time)

Enhanced Binary Search:
- One recursive search → $a = 1$
- Search half the remaining posts → $b = 2$
- Some extra processing per step → $d = 1$

Recipe: $a = 1,\ b = 2,\ d = 1 \rightarrow 1 < 2 \rightarrow$ Case 2

Result: Linear time — the search overhead dominates!

These examples show the Master Method applied to familiar technology problems like credit card processing, social media, and search features. Each example explains both the technical approach and the business impact, making the concepts more relatable and showing why algorithmic efficiency matters in real applications.

11 Practice & Application

11.1 Class Activity: Be the Algorithm Analyzer

Instructions: Work in pairs. For each real-world scenario, figure out a, b, d and predict the performance:

12 Scenarios

Scenario 1: Photo Editing App
You're building Instagram filters. To blur a photo:

- Split photo into **4 quadrants**, blur each separately
- Each quadrant has $\frac{1}{4}$ the pixels
- Combining quadrants is quick (proportional to total pixels)

Scenario 2: Video Game Graphics
Rendering 3D objects by breaking them down:

- Split complex 3D model into **16 simpler pieces**
- Each piece has $\frac{1}{4}$ the detail level
- Assembling final image takes time proportional to pixels^2

Scenario 3: Spotify Recommendations
Finding similar songs in a music database:

- Use **binary search** through sorted song list
- Each step eliminates **half** the remaining songs
- Significant processing needed at each step

Your Analysis*

For each scenario, determine:

1. How many subproblems? ($a = ?$)

2. How much smaller is each? ($b = ?$)

3. How much extra work? ($d = ?$)

4. Which case applies?

5. What's the final performance?

---

13 Solutions & Discussion

This activity uses familiar apps and scenarios that you likely interact with daily. The problems are presented as design challenges rather than abstract math problems (you're welcome!). Solutions include my commentary on practical implications, connecting the mathematical analysis back to real-world performance considerations.

13.1 When Our Algorithm Analyzer Doesn't Work

The Tool Has Limits: Our Algorithm Analyzer only works for certain types of problems

14 Requirements

Your algorithm must:
- Break problems into **equal-sized** pieces
- Always make the **same number** of recursive calls
- Do **roughly the same amount** of extra work each time

---

15 Problems We Can't Handle

Example 1: Cleaning Your Room
"Clean the biggest mess first, then clean everything else"
- Two subproblems but they're **different sizes**
- Our analyzer assumes equal sizes

Example 2: Planning a Trip
"Visit n cities, planning gets harder as trip gets longer"
- The planning work depends on trip length
- Our analyzer assumes **constant extra work**

Example 3: Processing Email
"Handle today's email, then yesterday's, then day before..."
- Reduces by **1 email per step**, not by a constant factor
- Our analyzer assumes you divide by a **fixed number**

---

16 What to Do Instead

- **Direct Analysis:** Count the steps manually
- **Other Tools:** Use more advanced mathematical techniques
- **Experimentation:** Run the algorithm and measure performance

---

This section explains the Master Method's limitations through relatable analogies rather than mathematical constraints. The examples should help you understand when the tool applies and when you need alternative approaches. The focus is on recognizing problem patterns rather than formal mathematical requirements.

16.1 Real-World Reality Check

17 Why Strassen Isn't Always Used

The Theory vs Reality Problem:

Strassen's Advantages:
- Faster for huge problems (like Netflix's full user base)
- Shows that *"impossible"* improvements are possible
- Inspired decades of further research

Strassen's Disadvantages:
- **Complex to implement** — easy to make bugs
- **Uses more memory** — lots of temporary storage
- **Only faster for big problems** — small problems run slower
- **Numerical errors** — subtractions can amplify rounding mistakes

---

18 What Companies Actually Do

Hybrid Approaches:
- **Small problems:** Use simple method (faster due to less overhead)
- **Large problems:** Switch to Strassen-style methods
- **Find the sweet spot:** Experiment to find best crossover point

Modern Examples:
- **Video games:** Simple method for small 3D objects, advanced for complex scenes
- **Netflix:** Simple method for individual recommendations, advanced for batch processing
- **Phone cameras:** Simple method for preview, advanced for final photo processing

---

19 The Algorithm Analyzer in Practice

What it's good for:
- **Quick comparisons** between algorithm approaches
- **Understanding** why some methods are better
- **Predicting** which approach will win for large problems

What it misses:
- Real hardware effects (cache, parallel processing)
- Implementation complexity and bugs
- Small problem performance

This section connects the theoretical analysis to practical engineering decisions. You learn that mathematical analysis is just one factor in choosing algorithms - implementation complexity, hardware considerations, and problem size all matter. The examples show how real companies make these tradeoffs.

19.1 Recap

The Matrix Multiplication Story

The Problem: Netflix, video games, and AI need to multiply huge grids of numbers fast
First Try: Use the obvious method → n^3 steps (too slow for big problems)
Second Try: Divide and conquer → still n^3 steps (divide-and-conquer alone isn't magic)
The Breakthrough: Strassen's 1969 discovery → reduce 8 calculations to 7 → $n^{2.807}$ steps
The Lesson: Sometimes *"impossible"* improvements are possible with creative thinking

The Algorithm Analyzer Tool (Master Method)

What it does: Predicts performance of any "divide and conquer" algorithm
How to use it:
1. Count recursive calls (a)
2. See how much smaller subproblems get (b)
3. Measure extra work (d)
4. Compare a with b^d to get your answer

Three outcomes: Balanced (Case 1), Top-heavy (Case 2), Bottom-heavy (Case 3)
When it helps: Quick analysis, comparing approaches, understanding why algorithms work

This summary emphasizes the real-world motivation and practical value of what you learned. The matrix multiplication story shows how algorithmic breakthroughs can solve practical problems, while the Algorithm Analyzer gives you a concrete tool for future algorithm analysis.

19.2 Final Check: Do You Get It?

Question 1:
Imagine someone found an even better way to do Strassen's algorithm with **only 6 recursive**

calls instead of 7.
What would happen to the performance?
- Hint: $a = 6,\ b = 2,\ d = 2\ \rightarrow\ n^{\log_2 6} \approx n^{2.585}$ — even faster than $n^{2.807}$.

Question 2:
Why didn't our first "divide and conquer" attempt improve over the basic method,
even though breaking problems into pieces usually helps?
- Hint: Splitting into equal halves but still doing **8 sub-multiplications** $\rightarrow a = 8,\ b = 2,\ d = 2$ $\rightarrow$ still n^3,
so overhead wasn't reduced.

Question 3:
You're building a **mobile app**.
When would you choose the **simple** n^3 method over Strassen's faster $n^{2.807}$ method?
- Hint: For **small matrices** or memory-limited devices — Strassen's overhead and extra memory can make it slower or impractical.

19.3 Answers

1. **Huge improvement!**

   Recipe: $a = 6,\ b = 2,\ d = 2\ \rightarrow\ 6 > 4$ Case 3 $\rightarrow n^{2.585}$ time (even faster!)

2. **Too many recursive calls!**

   We created **8 subproblems**, which overwhelmed any benefit from making them smaller.

3. **For small problems** (simpler code, fewer bugs), **when battery life matters** (simpler operations), or **when precision is critical** (fewer rounding errors).

Question 1 shows how further improvements compound. Question 2 tests the core insight about recursive calls. Question 3 connects to practical engineering decisions you might face in app development or other projects.

19.4 That's all!

Questions about anything we covered?

> **i** What's Next?
>
> - **Practice:** Try the Algorithm Analyzer on algorithms you encounter
> - **Explore:** Look up how your favorite apps handle large-scale computation
> - **Think:** Notice when apps seem fast or slow - often it's algorithm choice!
> - **Connect:** See how these ideas apply to data science, web development, or game

development