Chpater 4. Divide-and-Conquer

Joon Soo Yoo

March 20, 2025

Assignment

- Read §4.1, §4.2, §4.3
- Problems:
  - §4.1 - #4
  - §4.2 - #1, 2, 5
  - §4.3 - #1(a), 1(c), 2

Chapter 4: Divide-and-Conquer

- Chapter 4.1: Multiplying square matrices
- Chapter 4.2: Strassen's algorithm for matrix multiplication
- Chapter 4.3: The substitution method for solving recurrences
- Chapter 4.4: The recursion-tree method for solving recurrences
- Chapter 4.5: The master method for solving recurrences
- *Chapter 4.6: Proof of the continuous master theorem*
- *Chapter 4.7: Akra-Bazzi recurrences*

What is Divide-and-Conquer?

A powerful strategy for designing asymptotically efficient algorithms.

- We've already seen divide-and-conquer in **Merge Sort** (Section 2.3.1).
- The key idea: solve a problem recursively by breaking it into smaller pieces.
- This chapter focuses on algorithm design & solving recurrence relations.

Divide-and-Conquer Paradigm

A recursive algorithm with three main steps:

1. **Divide**: Break the problem into smaller subproblems.
2. **Conquer**: Solve each subproblem recursively.
3. **Combine**: Merge subproblem solutions into a global solution.

The recursion continues until reaching a **base case**, small enough to solve directly.

Why Recurrences?

Recurrences describe the running time of recursive algorithms.

- A **recurrence** is an equation for a function defined in terms of smaller inputs.
- Example: Merge Sort's worst-case recurrence from Section 2.3.2.
- Understanding recurrences helps analyze and design efficient algorithms.

What is an Algorithmic Recurrence?

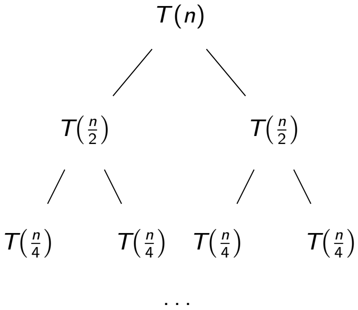
A recurrence $T(n)$ is *algorithmic* if:

1. $T(n) = \Theta(1)$ for all $n < n_0$ (base case threshold).
2. Every recursive path eventually reaches a base case.

These properties ensure the algorithm:

- Runs in finite time.
- Solves at least one valid input of each size $n < n_0$.

Merge Sort: Recursion Tree Intuition



Merge Sort Recurrence and Base Case

Recurrence:

$$T(n) \;=\; 2\,T\!\left(\tfrac{n}{2}\right) \;+\; \Theta(n), \quad \text{for } n > 1.$$

Base Case:

$$T(1) \;=\; \Theta(1).$$

Even though sorting a single element is trivial (no comparisons needed), we still incur a *constant cost*:

- Checking if $n = 1$.
- Possibly returning or copying the element.
- Function-call overhead, etc.

Solution: Recursion-tree analysis,

$$T(n) \;=\; \Theta(n \log n).$$

Recurrence Conventions

To simplify recurrence analysis:

- Assume recurrences are **algorithmic** unless stated otherwise.
- Ignore floors/ceilings when they don't affect asymptotics.
- If recurrence is an inequality:
  - Use $O(\cdot)$ for upper bounds (e.g., $T(n) \leq 2T(n/2) + \Theta(n)$).
  - Use $\Omega(\cdot)$ for lower bounds (e.g., $T(n) \geq 2T(n/2) + \Theta(n)$).

Examples of Recurrences

Different divide-and-conquer algorithms yield different recurrences:

- $T(n) = 8T(n/2) + \Theta(1) \to \Theta(n^3)$ (simple matrix multiplication)
- $T(n) = 7T(n/2) + \Theta(n^2) \to \Theta(n^{\log_2 7})$ (Strassen's algorithm)
- $T(n) = T(n/3) + T(2n/3) + \Theta(n) \to \Theta(n \log n)$
- $T(n) = T(n/5) + T(7n/10) + \Theta(n) \to \Theta(n)$ (Chapter 9)
- $T(n) = T(n-1) + \Theta(1) \to \Theta(n)$ (e.g., recursive linear search)

Coming Up: Tools to Solve Recurrences

We'll explore four methods for solving divide-and-conquer recurrences:

- **Substitution Method** (Section 4.3): Guess-and-prove via induction.
- **Recursion-Tree Method** (Section 4.4): Sum costs across recursion levels.
- **Master Method** (Sections 4.5–4.6): Fast asymptotic bounds for $T(n) = aT(n/b) + f(n)$.
- **Akra-Bazzi Method** (Section 4.7): Handles more general cases with calculus.

Chapter 4: Divide-and-Conquer

- **Chapter 4.1: Multiplying square matrices**
- Chapter 4.2: Strassen's algorithm for matrix multiplication
- Chapter 4.3: The substitution method for solving recurrences
- Chapter 4.4: The recursion-tree method for solving recurrences
- Chapter 4.5: The master method for solving recurrences
- *Chapter 4.6: Proof of the continuous master theorem*
- *Chapter 4.7: Akra-Bazzi recurrences*

Matrix Multiplication Basics

Let $A = (a_{ik})$ and $B = (b_{kj})$ be $n \times n$ matrices.

The (i, j)-th entry of $C = A \cdot B$ is:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{4.1}$$

Straightforward algorithm runs in $\Theta(n^3)$ time.

Straightforward Triple-Loop Algorithm

`MATRIX-MULTIPLY(A, B, C, n)`

- **For** $i = 1$ to n
-     **For** $j = 1$ to n
-         **For** $k = 1$ to n
-             $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

Runs in $\Theta(n^3)$ time.

Divide-and-Conquer Matrix Multiplication

Partition $n \times n$ matrices into $n/2 \times n/2$ submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Submatrix Multiplication

Compute $C = A \cdot B$ using:

$$
\begin{aligned}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{aligned}
$$

Requires 8 multiplications of $n/2 \times n/2$ matrices and 4 additions.

MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)

MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)

if $n == 1$
    // Base case.
    $c_{11} = c_{11} + a_{11} \cdot b_{11}$
    return
// Divide.
partition A, B, and C into $n/2 \times n/2$ submatrices
    $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$;
    and $C_{11}, C_{12}, C_{21}, C_{22}$; respectively
// Conquer.
MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{11}, C_{11}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{12}, C_{12}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{11}, C_{21}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{12}, C_{22}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{21}, C_{11}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{22}, C_{12}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{21}, C_{21}, n/2)$
MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{22}, C_{22}, n/2)$

Base Case and Partitioning

Base Case ($n = 1$):

- Each matrix is 1×1, so just a single scalar.
- Computation of $C \leftarrow C + A \times B$ involves:

  $$c_{11} \leftarrow c_{11} + a_{11} \cdot b_{11},$$

  i.e., one multiplication + one addition.
- Therefore, $T(1) = \Theta(1)$.

Partitioning Cost:

- Uses *index calculations* (no bulk copying).
- We just compute offsets to define $\frac{n}{2} \times \frac{n}{2}$ submatrices.
- Hence, partitioning is a constant-time $\Theta(1)$ step, *independent* of n.

Eight Subproblems and Overall Recurrence

Eight Subproblems:

- After partitioning, we have 8 recursive calls, each on $\frac{n}{2} \times \frac{n}{2}$ submatrices.
- Each contributes $T\left(\frac{n}{2}\right)$ to the running time.
- Only overhead: a $\Theta(1)$ partition step.

Time Complexity of Recursive Algorithm

Let $T(n)$ be the running time.

$$T(n) = 8T(n/2) + \Theta(1)$$

Using the Master Method (Section 4.5), we get:

$$T(n) = \Theta(n^3)$$

Same as the triple-loop algorithm — no speedup yet.

Why Not Faster Than Merge Sort?

Merge Sort:

$$T_{\text{merge}}(n) \;=\; 2\,T\!\left(\tfrac{n}{2}\right) \;+\; \Theta(n) \;\Longrightarrow\; \Theta(n\log n).$$

- Fewer subproblems per level (only 2).
- Combine step (merging) costs $\Theta(n)$ each level.

Recursive Matrix Multiply:

$$T(n) \;=\; 8\,T\!\left(\tfrac{n}{2}\right) \;+\; \Theta(1) \;\Longrightarrow\; \Theta(n^3).$$

- More subproblems per level (8).
- Combine step is cheap: $\Theta(1)$ per level.

Conclusion:

- Having only 2 subproblems yields $\Theta(n\log n)$.
- Having 8 subproblems, despite cheaper combine, grows to $\Theta(n^3)$.
- The large branching factor (8) outweighs the small $\Theta(1)$ combine.

Exercise 4.1-3 (CLRS 4th ed.)

Problem Statement:

"MATRIX-MULTIPLY-RECURSIVE" (page 83) partitions matrices A, B, C by *index calculation*, taking $\Theta(1)$ time. Suppose instead that you *copy* the appropriate elements of A, B, and C into separate $\frac{n}{2} \times \frac{n}{2}$ submatrices

$$A_{11}, A_{12}, A_{21}, A_{22}, \quad B_{11}, B_{12}, B_{21}, B_{22}, \quad C_{11}, C_{12}, C_{21}, C_{22}$$

respectively, and after the recursive calls, you copy the results from $C_{11}, C_{12}, C_{21}, C_{22}$ back into the appropriate places in C.

Question: How does the recurrence (4.9) change, and what is its solution?

Solution to Exercise 4.1-3

Copying Cost:

- Copying *all* relevant submatrices of A, B, and C (and then copying back) touches $\Theta(n^2)$ elements at each level of recursion.
- Thus, the partition+combine overhead is now $\Theta(n^2)$ instead of $\Theta(1)$.

New Recurrence:

$$T(n) = 8\,T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Chapter 4: Divide-and-Conquer

- Chapter 4.1: Multiplying square matrices
- **Chapter 4.2: Strassen's algorithm for matrix multiplication**
- Chapter 4.3: The substitution method for solving recurrences
- Chapter 4.4: The recursion-tree method for solving recurrences
- Chapter 4.5: The master method for solving recurrences
- *Chapter 4.6: Proof of the continuous master theorem*
- *Chapter 4.7: Akra-Bazzi recurrences*

4.2 Strassen's Algorithm for Matrix Multiplication

Matrix multiplication in less than $\Theta(n^3)$ time?

- Until 1969, many believed n^3 multiplications were necessary.
- V. Strassen proved a *remarkable* divide-and-conquer method.
- **Strassen's running time**: $\Theta\left(n^{\lg 7}\right) \approx \Theta(n^{2.81})$.
- This beats the classical $\Theta(n^3)$ approach.

Main idea:

- Same *recursive* partitioning of $n \times n$ matrices into $n/2 \times n/2$ blocks.
- But reduce the *number of matrix multiplications* from 8 down to 7.
- Pay a small overhead of extra additions (still only a constant factor).

How to Reduce Multiplications?

Motivation from a simple scalar example:

$$x^2 - y^2 \quad \text{(normally needs 2 multiplications)}$$

But recall the identity:

$$x^2 - y^2 = (x+y)\,(x-y),$$

which needs only *1 multiplication* $(x+y)\cdot(x-y)$, plus 2 additions.

Why helpful for matrices?

- For scalars, both methods cost 3 operations, so no big deal.
- For *large* matrices:
  - Multiplication is $\Theta(n^3)$ (classical).
  - Addition is only $\Theta(n^2)$.
- *Replacing one matrix multiplication with a few more additions* can lower total cost.

Strassen's Algorithm: Steps 1 and 2

Step 1: Base Case / Partition

- If $n = 1$, each matrix is just a single element.
  - Do 1 scalar multiply + 1 scalar add ($\Theta(1)$).
  - Return.
- Otherwise, partition A, B, and C each into four $\frac{n}{2} \times \frac{n}{2}$ submatrices.
- Partitioning by index calculation: $\Theta(1)$.

Step 2: Form S- and P- storage

- Create 10 *sum/difference* matrices $S_1, \ldots, S_{10}$ (each $\frac{n}{2} \times \frac{n}{2}$).
- Zero-initialize 7 product-result matrices $P_1, \ldots, P_7$.
- All told, 17 submatrices, each with $\frac{n}{2} \times \frac{n}{2}$ entries.
- Cost: $\Theta(n^2)$, since we write all these entries once.

Strassen's Step 2 and Step 3: S and P Matrices

Step 2: Construct 10 matrices $S_1, \ldots, S_{10}$ (each $n/2 \times n/2$)

$$
\begin{aligned}
&S_1 = B_{12} - B_{22}, \quad S_2 = A_{11} + A_{12}, \quad S_3 = A_{21} + A_{22}, \\
&S_4 = B_{21} - B_{11}, \quad S_5 = A_{11} + A_{22}, \quad S_6 = B_{11} + B_{22}, \\
&S_7 = A_{12} - A_{22}, \quad S_8 = B_{21} + B_{22}, \quad S_9 = A_{11} - A_{21}, \quad S_{10} = B_{11} + B_{12}.
\end{aligned}
$$

Step 3: Recursively compute 7 matrices $P_1, \ldots, P_7$ (each $n/2 \times n/2$)

$$
\begin{aligned}
P_1 &= A_{11} \times S_1, & &\text{(corresponds to } A_{11}(B_{12} - B_{22})), \\
P_2 &= S_2 \times B_{22}, & &((A_{11} + A_{12})B_{22}), \\
P_3 &= S_3 \times B_{11}, & &((A_{21} + A_{22})B_{11}), \\
P_4 &= A_{22} \times S_4, & &(A_{22}(B_{21} - B_{11})), \\
P_5 &= S_5 \times S_6, & &((A_{11} + A_{22})(B_{11} + B_{22})), \\
P_6 &= S_7 \times S_8, & &((A_{12} - A_{22})(B_{21} + B_{22})), \\
P_7 &= S_9 \times S_{10}, & &((A_{11} - A_{21})(B_{11} + B_{12})).
\end{aligned}
$$

Aligning the C_{11} Expansion (Strassen Step 4)

Update for C_{11}:

$$C_{11} \;\leftarrow\; C_{11} + P_5 + P_4 - P_2 + P_6.$$

Expanding each P_i:

$$\underbrace{(A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22})}_{P_5}$$

$$+ \underbrace{(A_{22}B_{22} - A_{22}B_{11})}_{P_4} - \underbrace{(A_{11}B_{22} + A_{12}B_{22})}_{P_2} + \underbrace{(A_{22}B_{22} - A_{22}B_{21} + A_{12}B_{22} + A_{12}B_{21})}_{P_6}$$

$$= A_{11}B_{11} + A_{12}B_{21}.$$

Observation: Notice how many terms subtract out in the middle lines. Everything simplifies to exactly $A_{11}B_{11} + A_{12}B_{21}$, which matches the formula for $C_{11} = A_{11}B_{11} + A_{12}B_{21}$.

Final Updates for C_{12}, C_{21}, and C_{22}

1. $C_{12} \leftarrow C_{12} + P_1 + P_2$

$$\begin{aligned} &(A_{11}B_{12} - A_{11}B_{22}) \quad + \quad ((A_{11} + A_{12})B_{22}) \\ &= A_{11}B_{12} + A_{12}B_{22}. \end{aligned}$$

Hence, $C_{12} = A_{11}B_{12} + A_{12}B_{22}$.

2. $C_{21} \leftarrow C_{21} + P_3 + P_4$

$$\begin{aligned} &((A_{21} + A_{22})B_{11}) \quad + \quad (A_{22}(B_{21} - B_{11})) \\ &= A_{21}B_{11} + A_{22}B_{21}. \end{aligned}$$

Hence, $C_{21} = A_{21}B_{11} + A_{22}B_{21}$.

3. $C_{22} \leftarrow C_{22} + P_5 + P_1 - P_3 - P_7$
(omitting intermediate lines)

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Strassen's Algorithm: Complexity Summary

Step 1: Base Case / Partition ($\Theta(1)$ cost)

- If $n = 1$, just 1 scalar multiply + 1 add $\Rightarrow \Theta(1)$.
- Otherwise, partition into $\frac{n}{2} \times \frac{n}{2}$ submatrices, also in $\Theta(1)$ time by index calculation.

Step 2: Create S_i and P_i storage ($\Theta(n^2)$ cost)

- 10 sums/differences $S_1, \ldots, S_{10}$, plus 7 zero-initialized P_i.
- Touches n^2 elements total $\implies \Theta(n^2)$.

Step 3: 7 Recursive Multiplications

$$\text{Cost} = 7\,T\Big(\frac{n}{2}\Big).$$

(Instead of 8, saving one multiplication in exchange for extra adds.)

Step 4: Combine Results ($\Theta(n^2)$ cost)

- Add/subtract P_i's into C_{ij} submatrices $\implies \Theta(n^2)$.

Overall Recurrence:

$$T(n) \;=\; 7\,T\Big(\frac{n}{2}\Big) \;+\; \Theta(n^2).$$

Master Method yields $T(n) = \Theta\big(n^{\log_2 7}\big) \approx \Theta(n^{2.81})$.

Comparing Naive Divide-and-Conquer vs. Strassen

Naive Divide-and-Conquer:

$$T_{\text{naive}}(n) \ = \ 8\,T\left(\tfrac{n}{2}\right) \ + \ \Theta(1).$$

- $\Rightarrow T_{\text{naive}}(n) = \Theta\left(n^3\right)$.
- 8 subproblems at each level; partition/combine is cheap ($\Theta(1)$).

Strassen's Method:

$$T_{\text{Strassen}}(n) \ = \ 7\,T\left(\tfrac{n}{2}\right) \ + \ \Theta(n^2).$$

- $\Rightarrow T_{\text{Strassen}}(n) = \Theta\left(n^{\log_2(7)}\right) \approx \Theta(n^{2.81})$.
- Fewer recursive multiplications (7 vs. 8), but more $\Theta(n^2)$ addition overhead.

Conclusion:

- Strassen's clever trade-off yields a strictly faster algorithm *asymptotically*.
- Both beat the triple-loop $\Theta(n^3)$ approach, but Strassen's outperforms naive DC.

Chapter 4: Divide-and-Conquer

- Chapter 4.1: Multiplying square matrices
- Chapter 4.2: Strassen's algorithm for matrix multiplication
- **Chapter 4.3: The substitution method for solving recurrences**
- Chapter 4.4: The recursion-tree method for solving recurrences
- Chapter 4.5: The master method for solving recurrences
- *Chapter 4.6: Proof of the continuous master theorem*
- *Chapter 4.7: Akra-Bazzi recurrences*

4.3 The Substitution Method: Intuition and Setup

The substitution method = guess + induction.

Step 1: Make an educated guess.
We think the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$$

might be bounded by $T(n) = O(n \log n)$.

Step 2: Try to prove it by induction.
We assume the bound holds for *smaller values*:

$$\text{Assume } T(k) \le c\, k \log k \quad \text{for all } n_0 \le k < n.$$

(Note: the recurrence only needs $T(\lfloor n/2 \rfloor)$, but induction assumes all smaller k.)

Why this assumption?

- We're trying to show $T(n)$ doesn't grow faster than $n \log n$.
- This assumption lets us plug into the recurrence and test if our guess works.
- If it fails, we adjust constants (c, n_0) until it holds.

This is how we "substitute" our guess into the recurrence to prove the bound!

Step 2: Substitution into the Recurrence

Assume $n \geq 2n_0$, so the inductive hypothesis applies to $\lfloor n/2 \rfloor$.

Apply the recurrence:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$$

Apply the inductive hypothesis:

$$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$$

Approximate:

$$\leq c \frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{c\,n}{2}(\log n - 1)$$

Now plug in:

$$T(n) \leq 2 \cdot \frac{c\,n}{2}(\log n - 1) + a\,n = c\,n \log n - c\,n + a\,n = c\,n \log n + (a - c)n$$

Step 3: Choosing Constants

We want:

$$T(n) \leq c\, n \log n$$

So require:

$$(a - c)n \leq 0 \quad \Rightarrow \quad c \geq a$$

Therefore:

- Choose $c \geq a$ (where a comes from the hidden constant in $\Theta(n)$)
- Then for all $n \geq 2n_0$, we have:

$$T(n) \leq c\, n \log n$$

Substitution Method: Verifying Base Cases

We showed the inductive step holds for $n \geq 2n_0$.

Now verify the base cases: $n_0 \leq n < 2n_0$.
Choose $n_0 = 2 \Rightarrow n = 2, 3$.

- Since $T(n)$ is algorithmic, $T(2)$ and $T(3)$ are constant.
- Set $c = \max\{T(2), T(3)\}$
- Then:

$$T(2) \leq c < 2c \log 2, \quad T(3) \leq c < 3c \log 3$$

Conclusion:

$$T(n) \leq c\, n \log n \quad \text{for all } n \geq 2 \;\Rightarrow\; T(n) = O(n \log n)$$

Practical Notes on Substitution Method (Base Cases)

Do we always need to write out the base case proof in full?

Not usually. In the algorithms literature:

- People rarely show full base case details.
- Most divide-and-conquer recurrences bottom out when n is small.
- It's standard to assume:

$$T(n) \le c\, n \log n \quad \text{for } n_0 \le n < n_0'$$

  for some constants $n_0,\ n_0' > n_0$ (e.g., $n_0' = 2n_0$).
- Then we just choose c large enough to make the inequality hold.

Conclusion: The base case is usually routine. Once the inductive step is proven, the full proof is considered complete.

Making a Good Guess for Substitution

How do you guess a solution to a recurrence?

- **No general formula** — it takes intuition, experience, and practice.
- If a recurrence *resembles* a known one, guess similarly:

$$T(n) = 2T(n/2 + 17) + \Theta(n) \;\Rightarrow\; O(n \log n)$$

  (because "+17" doesn't matter asymptotically)
- Use **bounding technique:**
  - Start with a rough range:

$$\Omega(n) \leq T(n) \leq O(n^2)$$

  - Refine both bounds until they meet at $\Theta(n \log n)$.

Bottom line: Guess, test, refine — and develop your recurrence intuition!

Trick of the Trade: Subtracting a Low-Order Term

Problem: Trying to prove $T(n) = O(n)$ for

$$T(n) = 2T(n/2) + \Theta(1)$$

Guessing $T(n) \le c\,n$ doesn't work:

$$T(n) \le 2 \cdot c(n/2) + a = c\,n + a \quad \not\le \quad c\,n$$

Fix: Strengthen the inductive hypothesis:

$$T(n) \le c\,n - d \quad \text{for some } d > 0$$

Now:

$$T(n) \le 2\left(\frac{c\,n}{2} - d\right) + a = c\,n - 2d + a$$

Choose $d > a$:

$$T(n) \le c\,n - (\text{something positive}) \;\Rightarrow\; T(n) \le c\,n$$

Insight: Subtracting a small term once for each recursive call helps the inequality close.

Avoiding Pitfalls in the Substitution Method

Don't use asymptotic notation in the inductive hypothesis!

Wrong approach:

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$$

$$\text{Assume } T(n) = O(n) \Rightarrow T(n) \le 2 \cdot O(n) + \Theta(n) = O(n)$$

Why it's wrong:

- The constants inside $O(n)$ and $\Theta(n)$ are hidden and may vary.
- You lose control over the exact bound required for induction.
- Can't conclude $T(n) \le c\, n$ from vague $O(n)$ steps.

Correct approach:

- Explicitly guess $T(n) \le c\, n$
- Carry out the proof with real constants.
- At the end, conclude $T(n) = O(n)$ once all constants are handled.

Question?