Recursive Algorithms

Understanding Recursion and Recurrence Relations

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

Today's Roadmap

Concepts

1. What is divide & conquer?
2. The three-part template
3. When does it work?

Examples worked in full

4. Binary search
5. Merge sort
6. Karatsuba multiplication

Solving recurrences

7. Method 1: Unrolling
8. Method 2: Recursion tree
9. Method 3: Master Theorem

The One-Sentence Definition

Definition

Divide the problem into smaller subproblems of the same type,
conquer each recursively, then combine the results.

The key insight — return to this after every example

- The recursive structure of your algorithm gives you a recurrence relation **for free**.
- Solving that relation tells you the algorithm's cost.

$$T(n) = a\,T(n/b) + f(n)$$

The Three-Part Template

1. Divide

- Split the input into subproblems.
- Ideally: equal-sized pieces.
- Cost of splitting $\rightarrow$ part of $f(n)$.

2. Conquer

- Solve each subproblem recursively.
- Base case: when n is small, solve directly.
- Each call $\rightarrow$ the $aT(n/b)$ term.

3. Combine

- Merge subproblem answers into a solution.
- This is often the expensive step.
- Cost of merging $\rightarrow$ part of $f(n)$.

Ask yourself

Before we look at any example: which step do you think does the most work?

(Answer varies — that's the point. We'll see all three cases today.)

When Does Divide & Conquer Help?

Ask these four questions before writing a line of code:

1. Can I split the problem into independent (or near-independent) subproblems?
2. Are the subproblems the same kind of problem as the original?
3. How much does it cost to split? How much to combine?
4. Do I need both halves, or can I discard one?

Key distinction

Question 4 separates **divide and conquer** (use all subproblems) from **decrease and conquer** (discard one, recurse into the other).

Binary search is decrease and conquer — but we still call it D&C loosely.

What is Recursion?

A function that calls itself

Recursion: A function that solves a problem by calling itself on smaller versions

Simple example - Factorial:

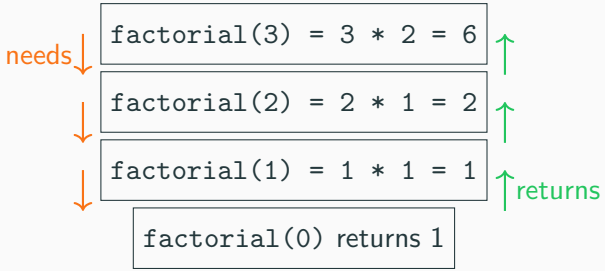
```
int factorial(int n) {
    if (n == 0) return 1;          // base case
    return n * factorial(n - 1); // recursive call
}
```

Key components:

1. **Base case:** When to stop recursing
2. **Recursive case:** How to reduce the problem
3. **Progress:** Each call must get closer to base case

How Recursion Works: The Call Stack

Example: factorial(3)



Each call waits for the one below it to return

Recursion vs Iteration: Same Problem, Different Approaches

Recursive factorial:

```
int factorial(int n) {
  if (n == 0) return 1;
  return n * factorial(n-1);
}
```

Pros:

- Elegant, readable
- Natural for some problems

Cons:

- Function call overhead
- Stack space usage

Iterative factorial:

```
int factorial(int n) {
  int result = 1;
  for (int i = 1; i <= n; i++)
      {
    result *= i;
  }
  return result;
}
```

Pros:

- No call overhead
- Constant space

Cons:

- Sometimes less intuitive

From Recursive Code to Recurrence Relations

How do we analyze recursive algorithms?

For iterative code: We count loop iterations

For recursive code: We write a recurrence relation

Recurrence relation: Expresses $T(n)$ in terms of T on smaller inputs

General form:

- $T(n)$ = time to solve problem of size n
- Express $T(n)$ using:
  - Time for recursive calls on smaller problems
  - Time for non-recursive work
- Specify base case(s)

Example: Factorial Recurrence

```
int factorial(int n) {
    if (n == 0) return 1;          // O(1)
    return n * factorial(n - 1); // O(1) + T(n-1)
}
```

Analysis:

- Base case: $T(0) = O(1)$
- Recursive case:
  - One recursive call: $T(n-1)$
  - Plus constant work (multiplication): $O(1)$

Recurrence:

$$T(0) = c_1$$
$$T(n) = T(n-1) + c_2 \quad \text{for } n > 0$$

Solving by Expansion

Factorial: $T(n) = T(n-1) + c$

Method: Expand until we see a pattern

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= [T(n-2) + c] + c \\ &= T(n-2) + 2c \\ &= [T(n-3) + c] + 2c \\ &= T(n-3) + 3c \\ &\vdots \\ &= T(n-k) + kc \end{aligned}$$

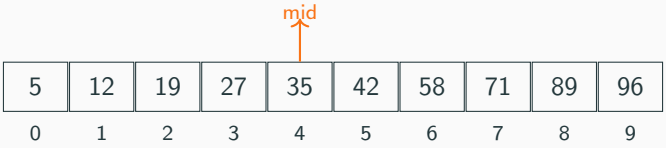
Stop when we hit base case:
$n - k = 0$, so $k = n$

$$\begin{aligned} T(n) &= T(0) + nc \\ &= c_1 + nc \\ &= O(n) \end{aligned}$$

Result: $\Theta(n)$

Binary Search — The Idea

Problem: Find target value **42** in a sorted array.



Divide: Find the middle index

$$\texttt{mid} = \lfloor (\texttt{lo} + \texttt{hi})/2 \rfloor$$

Conquer: Is `arr[mid]` $==$ target?

- If $<$ target, recurse right half
- If $>$ target, recurse left half

Combine: Nothing to do. The recursive answer *is* the answer.

Note: No combine step! This is **decrease & conquer.**

Binary Search — Full Worked Example

Array: [5, 12, 19, 27, 35, 42, 58, 71, 89, 96] **Target:** 42

Step-by-step

- **Step 1:** `lo=0, hi=9` `mid=4` `arr[4]=35`
  $35 < 42 \Rightarrow$ search right half (`lo=5, hi=9`)

Binary Search — Full Worked Example

Array: [5, 12, 19, 27, 35, 42, 58, 71, 89, 96] **Target:** 42

Step-by-step

- **Step 1:** `lo=0, hi=9  mid=4  arr[4]=35`
  $35 < 42 \Rightarrow$ search right half (`lo=5, hi=9`)
- **Step 2:** `lo=5, hi=9  mid=7  arr[7]=71`
  $71 > 42 \Rightarrow$ search left half (`lo=5, hi=6`)

Binary Search — Full Worked Example

Array: [5, 12, 19, 27, 35, 42, 58, 71, 89, 96] **Target:** 42

Step-by-step

- **Step 1:** `lo=0, hi=9 mid=4 arr[4]=35`
  $35 < 42 \Rightarrow$ search right half (`lo=5, hi=9`)
- **Step 2:** `lo=5, hi=9 mid=7 arr[7]=71`
  $71 > 42 \Rightarrow$ search left half (`lo=5, hi=6`)
- **Step 3:** `lo=5, hi=6 mid=5 arr[5]=42`
  $42 = 42 \Rightarrow$ found at index **5**

✓ **3 comparisons to search 10 elements**

Ask yourself

- How many comparisons would linear search need in the worst case?

Binary Search — Full Worked Example

Array: [5, 12, 19, 27, 35, 42, 58, 71, 89, 96] **Target:** 42

Step-by-step

- **Step 1:** `lo=0, hi=9 mid=4 arr[4]=35`
  $35 < 42 \Rightarrow$ search right half (`lo=5, hi=9`)
- **Step 2:** `lo=5, hi=9 mid=7 arr[7]=71`
  $71 > 42 \Rightarrow$ search left half (`lo=5, hi=6`)
- **Step 3:** `lo=5, hi=6 mid=5 arr[5]=42`
  $42 = 42 \Rightarrow$ found at index **5**

✓ **3 comparisons to search 10 elements**

Ask yourself

- How many comparisons would linear search need in the worst case? **Answer: 10.**
- How many would binary search need in the worst case on 10 elements?

Binary Search — Full Worked Example

Array: [5, 12, 19, 27, 35, 42, 58, 71, 89, 96] **Target:** 42

Step-by-step

- **Step 1:** `lo=0, hi=9 mid=4 arr[4]=35`
  $35 < 42 \Rightarrow$ search right half (`lo=5, hi=9`)
- **Step 2:** `lo=5, hi=9 mid=7 arr[7]=71`
  $71 > 42 \Rightarrow$ search left half (`lo=5, hi=6`)
- **Step 3:** `lo=5, hi=6 mid=5 arr[5]=42`
  $42 = 42 \Rightarrow$ found at index **5**

✓ **3 comparisons to search 10 elements**

Ask yourself

- How many comparisons would linear search need in the worst case? **Answer: 10.**
- How many would binary search need in the worst case on 10 elements? **Answer: 4, since** $\lceil \log_2 10 \rceil = 4$**.**

Binary Search — Recurrence and Solution

Write the recurrence:

$$T(1) = c_0 \qquad T(n) = T(n/2) + c$$

Interpretation

- $T(n/2)$: one recursive call on half the array
- c: constant work (compute `mid`, compare)

Solve by unrolling

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + c + c \\ &= T(n/8) + 3c \\ &\vdots \\ &= T(n/2^k) + kc \end{aligned}$$

Stop when $n/2^k = 1 \Rightarrow k = \log_2 n$:

$$T(n) = T(1) + c \log_2 n = \Theta(\log n)$$

Each step halves the problem. Starting from n, we can halve $\log_2 n$ times before reaching 1.

Example: Binary Search

```
int binarySearch(int arr[], int left, int right, int target) {
    if (left > right) return -1;  // base case

    int mid = (left + right) / 2;
    if (arr[mid] == target) return mid;
    if (arr[mid] > target)
        return binarySearch(arr, left, mid-1, target);
    else
        return binarySearch(arr, mid+1, right, target);
}
```

Analysis:

- Search space halved each time
- One recursive call on half the array
- Constant work (comparisons)

Binary Search Recurrence

Recurrence:

$$T(1) = c$$
$$T(n) = T(n/2) + c \quad \text{for } n > 1$$

Solving by expansion:

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= [T(n/4) + c] + c = T(n/4) + 2c \\ &= [T(n/8) + c] + 2c = T(n/8) + 3c \\ &\vdots \\ &= T(n/2^k) + kc \end{aligned}$$

Base case when: $n/2^k = 1$, so $k = \log_2 n$

$$T(n) = T(1) + c \log n = O(\log n) = \Theta(\log n)$$

Insertion Sort

Insertion Sort Idea: Build a sorted prefix one element at a time.

Example: [38, 27, 43, 3, 9]

- Insert 27 into [38] $\rightarrow$ [27, 38]
- Insert 43 $\rightarrow$ [27, 38, 43]
- Insert 3 $\rightarrow$ shift 43, 38, 27 $\rightarrow$ [3, 27, 38, 43]
- Insert 9 $\rightarrow$ shift 43, 38, 27 $\rightarrow$ [3, 9, 27, 38, 43]

Worst-case behavior: For each element, we may shift almost the entire prefix.

$$T(n) = 1 + 2 + 3 + \cdots + (n-1)$$

Insertion Sort

Insertion Sort Idea: Build a sorted prefix one element at a time.

Example: [38, 27, 43, 3, 9]

- Insert 27 into [38] $\rightarrow$ [27, 38]
- Insert 43 $\rightarrow$ [27, 38, 43]
- Insert 3 $\rightarrow$ shift 43, 38, 27 $\rightarrow$ [3, 27, 38, 43]
- Insert 9 $\rightarrow$ shift 43, 38, 27 $\rightarrow$ [3, 9, 27, 38, 43]

Worst-case behavior: For each element, we may shift almost the entire prefix.

$$T(n) = 1 + 2 + 3 + \cdots + (n-1)$$

$$T(n) = \frac{n(n-1)}{2} = \Theta(n^2)$$

Contrast with Merge Sort:

$$\Theta(n^2) \quad \text{vs} \quad \Theta(n \log n)$$

For large n, $n \log n$ grows dramatically slower than n^2.

Merge Sort — The Idea

Problem: Sort the array **[38, 27, 43, 3, 9, 82, 10]**

Divide

- Split array in half.
- Left: **[38, 27, 43, 3]**
- Right: **[9, 82, 10]**
- Cost: $O(1)$ (find midpoint)

Conquer

- Recursively sort each half.
- Left $\rightarrow$ **[3, 27, 38, 43]**
- Right $\rightarrow$ **[9, 10, 82]**
- Two calls: $2T(n/2)$

Combine

- Merge two sorted halves.
- Result: **[3, 9, 10, 27, 38, 43, 82]**
- Cost: $O(n)$ (this is the work!)

Ask yourself

- Contrast with binary search: here we need **both halves**, and we **do have a combine**.

Merge Sort — The Idea

Problem: Sort the array **[38, 27, 43, 3, 9, 82, 10]**

Divide

- Split array in half.
- Left: **[38, 27, 43, 3]**
- Right: **[9, 82, 10]**
- Cost: $O(1)$ (find midpoint)

Conquer

- Recursively sort each half.
- Left $\rightarrow$ **[3, 27, 38, 43]**
- Right $\rightarrow$ **[9, 10, 82]**
- Two calls: $2T(n/2)$

Combine

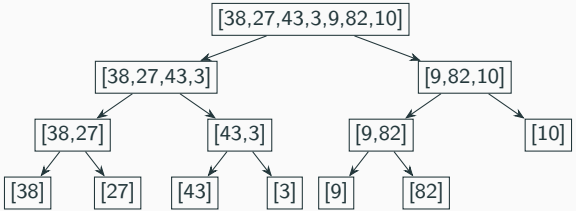
- Merge two sorted halves.
- Result: **[3, 9, 10, 27, 38, 43, 82]**
- Cost: $O(n)$ (this is the work!)

Ask yourself

- Contrast with binary search: here we need **both halves**, and we **do have a combine**.
- **Answer:** merge sort is true divide & conquer (use all subproblems) and the merge step is where we pay linear work.

Merge Sort — Full Worked Example (Recursive Calls)

Start: [38, 27, 43, 3, 9, 82, 10]



What recursion is doing:

- Split until size 1.
- Then merge on the way back up.

Merge Sort — What `mergeSort` Calls (and Why)

```
void mergeSort(int a[], int left, int right) {
  if (left >= right) return;          // base: size 1
  int mid = (left + right) / 2;       // divide
  mergeSort(a, left, mid);            // conquer left half
  mergeSort(a, mid+1, right);         // conquer right half
  merge(a, left, mid, right);         // combine (linear work)
}
```

Key point: The recursive calls do *not* directly sort the whole array. They sort the halves, then `merge` does the final work to combine them.

So the runtime is driven by:

$$\underbrace{2\,T(n/2)}_{\text{two halves}} + \underbrace{O(n)}_{\text{merge}}$$

Merge Sort — The `merge` Subroutine (Step-by-Step)

Goal: Merge two sorted halves into one sorted output.

Example merge:

$$\text{Left } [3, 27, 38, 43] \quad \text{and} \quad \text{Right } [9, 10, 82] \quad \longrightarrow \quad [3, 9, 10, 27, 38, 43, 82]$$

Idea: Two pointers i (left) and j (right). Repeatedly take the smaller.

Step	Compare	Output so far	Advance
1	$3 < 9$	$[3]$	$i{+}{+}$
2	$27 > 9$	$[3, 9]$	$j{+}{+}$
3	$27 > 10$	$[3, 9, 10]$	$j{+}{+}$
4	$27 < 82$	$[3, 9, 10, 27]$	$i{+}{+}$
5	$38 < 82$	$[3, 9, 10, 27, 38]$	$i{+}{+}$
6	$43 < 82$	$[3, 9, 10, 27, 38, 43]$	$i{+}{+}$
7	left empty	$[3, 9, 10, 27, 38, 43, 82]$	copy rest of right

Merge Sort — Recurrence and Solution

Write the recurrence:

Recurrence

$$T(1) = c_0 \qquad T(n) = 2T(n/2) + cn$$

Interpretation

- $2T(n/2)$: sort left + sort right
- cn: merge two sorted halves (linear)
- Depth is $\log_2 n$ (keep halving)

Recursion-tree intuition

- Level 0 merge work: cn
- Level 1 merge work: $2 \cdot c(n/2) = cn$
- Level 2 merge work: $4 \cdot c(n/4) = cn$
- $\cdots$

Merge Sort — Recurrence and Solution

Write the recurrence:

Recurrence

$$T(1) = c_0 \qquad T(n) = 2T(n/2) + cn$$

Interpretation

- $2T(n/2)$: sort left + sort right
- cn: merge two sorted halves (linear)
- Depth is $\log_2 n$ (keep halving)

Recursion-tree intuition

- Level 0 merge work: cn
- Level 1 merge work: $2 \cdot c(n/2) = cn$
- Level 2 merge work: $4 \cdot c(n/4) = cn$
- $\cdots$

Total work: each level costs cn, and there are $\log_2 n$ levels:

$$T(n) = cn \cdot \log_2 n + O(n) = \Theta(n \log n)$$

Key contrast with binary search: binary search has $T(n) = T(n/2) + O(1)$; merge sort has **two** subproblems **and** a linear combine.

Divide and Conquer: Merge Sort

Merge Sort Strategy:

1. **Divide:** Split array in half
2. **Conquer:** Recursively sort each half
3. **Combine:** Merge the two sorted halves

```
void mergeSort(int arr[], int left, int right) {
    if (left >= right) return;  // base case

    int mid = (left + right) / 2;
    mergeSort(arr, left, mid);       // sort left half
    mergeSort(arr, mid+1, right);    // sort right half
    merge(arr, left, mid, right);    // merge them
}
```

Key insight: Two recursive calls, each on half the array

Merge Sort Recurrence

Analysis:

- Two recursive calls, each on $n/2$ elements: $2T(n/2)$
- Merging takes $O(n)$ time

Recurrence:

$$T(1) = c$$
$$T(n) = 2T(n/2) + cn \quad \text{for } n > 1$$

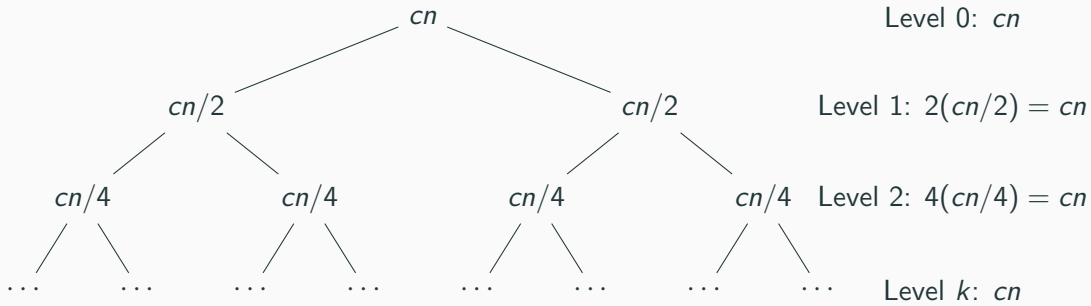
This is more complex! We need a systematic way to solve it.

Two methods:

1. Recursion tree (visual)
2. Master Theorem (formula)

Recursion Tree for Merge Sort

Visualizing the work at each level



Each level does cn work

How many levels? $\log_2 n$ (halving until we reach size 1)

Total: $cn \times \log_2 n = O(n \log n)$

Common Recurrence Patterns

Recurrence	Solution	Example
$T(n) = T(n-1) + c$	$O(n)$	Factorial, linear search
$T(n) = T(n/2) + c$	$O(\log n)$	Binary search
$T(n) = 2T(n/2) + cn$	$O(n \log n)$	Merge sort
$T(n) = 2T(n/2) + c$	$O(n)$	Tree traversal
$T(n) = T(n-1) + cn$	$O(n^2)$	Selection sort
$T(n) = 2T(n-1) + c$	$O(2^n)$	Towers of Hanoi

Pattern recognition is useful, but we need a general method...

Karatsuba — A Magic Trick First

Don't worry if this seems inscrutable. That's the point.

We want to compute 5678×1234. Name the four halves:
$a = 56,\ b = 78,\ c = 12,\ d = 34.$

Step 1:	$a \times c$	$= 56 \times 12 = 672$
Step 2:	$b \times d$	$= 78 \times 34 = 2652$
Step 3:	$(a+b) \times (c+d)$	$= 134 \times 46 = 6164$
Step 4:	$6164 - 672 - 2652$	$= \mathbf{2840}$

Combine (pad steps 1 and 4 with zeros):

$$\underbrace{672}_{\text{step 1}} \cdot 10^4 \ + \ \underbrace{2840}_{\text{step 4}} \cdot 10^2 \ + \ \underbrace{2652}_{\text{step 2}} \ = \ 6720000 + 284000 + 2652 \ = \ \mathbf{7{,}006{,}652}$$

Karatsuba — A Magic Trick First

Don't worry if this seems inscrutable. That's the point.

We want to compute 5678×1234. Name the four halves:
$a = 56,\ b = 78,\ c = 12,\ d = 34.$

Step 1:	$a \times c$	$= 56 \times 12 = 672$
Step 2:	$b \times d$	$= 78 \times 34 = 2652$
Step 3:	$(a+b) \times (c+d)$	$= 134 \times 46 = 6164$
Step 4:	$6164 - 672 - 2652$	$= \mathbf{2840}$

Combine (pad steps 1 and 4 with zeros):

$$\underbrace{672}_{\text{step 1}} \cdot 10^4 \ + \ \underbrace{2840}_{\text{step 4}} \cdot 10^2 \ + \ \underbrace{2652}_{\text{step 2}} \ = \ 6720000 + 284000 + 2652 \ = \ \mathbf{7{,}006{,}652}$$

You should have no intuition for why that worked. That's the point.

Karatsuba — The Recursive Structure

Why splitting in half suggests recursion

Any n-digit number can be written in terms of its two halves:

$$x = a \cdot 10^{n/2} + b \qquad y = c \cdot 10^{n/2} + d$$

In our example: $x = 5678,\ y = 1234,\ n = 4,\ a = 56,\ b = 78,\ c = 12,\ d = 34.$

Multiply out:

$$x \cdot y \ = \ ac \cdot 10^{n} \ + \ \underbrace{(ad + bc)}_{\text{cross term}} \cdot 10^{n/2} \ + \ bd \qquad (\star)$$

Straightforward plan: recursively compute ac, ad, bc, bd — four calls on $n/2$-digit numbers:

$$T(n) = 4\,T(n/2) + O(n) \ \Rightarrow \ \Theta(n^2)$$

No better than grade school. But look more carefully at $(\star)\ldots$

Karatsuba — Three Quantities, Not Four

The Gauss trick

Look at ($\star$) again. There are only **three** quantities we care about: ac, bd, $ad + bc$. We never need ad and bc separately — only their *sum*.

Can we get $ad + bc$ with one recursive call instead of two?

Karatsuba — Three Quantities, Not Four

The Gauss trick

Look at ($\star$) again. There are only **three** quantities we care about: ac, bd, $ad + bc$. We never need ad and bc separately — only their *sum*.

Can we get $ad + bc$ with one recursive call instead of two?

Yes. Compute $(a + b)(c + d)$ recursively and expand:

$$(a + b)(c + d) = ac + \mathbf{ad} + \mathbf{bc} + bd$$

Subtract steps 1 and 2:

$$(a + b)(c + d) \; - \; ac \; - \; bd \; = \; ad + bc \quad \checkmark$$

A trick due to Gauss (early 19th century). Three recursive calls total, plus $O(n)$ additions and shifts.

Karatsuba — Recurrence and Payoff

$$T(n) = 3T(n/2) + O(n)$$

Master Theorem: $a = 3,\ b = 2,\ n^{\log_2 3} \approx n^{1.585},\ f(n) = O(n) = O(n^{1.585-\varepsilon})$Case 1:

$$T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$$

	Naive recursive	Karatsuba
Recursive calls	4	3
Recurrence	$4T(n/2) + O(n)$	$3T(n/2) + O(n)$
Complexity	$\Theta(n^2)$	$\Theta(n^{1.585})$

Saving one recursive call per level breaks out of $O(n^2)$ entirely.

Aside: Strassen's Matrix Multiplication

The same idea, one level up

Naive matrix multiplication ($n \times n$ matrices): $\Theta(n^3)$.

Strassen (1969): divide each matrix into four $n/2 \times n/2$ blocks. Naive block multiply needs 8 recursive multiplications $\Rightarrow \Theta(n^3)$. Strassen reduces this to **7** via a similar algebraic trick:

$$T(n) = 7\,T(n/2) + O(n^2) \;\Rightarrow\; \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$$

Same moral: one fewer recursive call per level $\Rightarrow$ better exponent.

We won't work through the algebra — but the recurrence analysis is identical in structure to Karatsuba.

The Master Method

A black box for divide-and-conquer recurrences

What it does: you give it three numbers describing your recursive algorithm, it gives back the running time. No tree drawing required.

The format it requires: $T(n) = a\,T(n/b) + O(n^d)$

- a — number of recursive calls (e.g. merge sort: $a = 2$)
- b — factor by which input size shrinks (merge sort: $b = 2$)
- d — exponent of work *outside* recursive calls (merge sort: $d = 1$)

a, b, d are **constants** independent of n.

One assumption: all subproblems must be the *same size* (n/b). If subproblems have unequal sizes (e.g. $n/3$ and $2n/3$), the master method does not apply.

The Master Method — Three Cases

The trigger: compare a to b^d

Given $T(n) = aT(n/b) + O(n^d)$, the answer depends on one comparison:

$$a \quad \text{vs} \quad b^d$$

	Condition	Result	Intuition
Case 1	$a = b^d$	$O(n^d \log n)$	equal work every level
Case 2	$a < b^d$	$O(n^d)$	combine step dominates
Case 3	$a > b^d$	$O(n^{\log_b a})$	leaves dominate

The log in Case 1 needs no base: any base differs by a constant, absorbed by Big-O. The base b in Case 3's *exponent* matters — it is exactly the shrink factor.

Why Three Cases? The Recursion Tree Intuition

At level j of the recursion tree: a^j subproblems of size n/b^j, so work at level j is

$$a^j \cdot O\Big((n/b^j)^d\Big) = O(n^d) \cdot \big(\tfrac{a}{b^d}\big)^j$$

The ratio a/b^d tells you whether work grows, shrinks, or stays flat as you go deeper:

a/b^d	Work per level	Who wins
$= 1$	constant	all levels equal $\Rightarrow \times \log n$
< 1	shrinks	root dominates $\Rightarrow O(n^d)$
> 1	grows	leaves dominate $\Rightarrow O(n^{\log_b a})$

Leaf count: $a^{\log_b n} = n^{\log_b a}$ — that's the Case 3 exponent.

Master Method: Example 1 — Merge Sort

A sanity check on something we already know

Recurrence: $T(n) = 2T(n/2) + O(n)$

Step 1: identify a, b, d.

- How many recursive calls?

Master Method: Example 1 — Merge Sort

A sanity check on something we already know

Recurrence: $T(n) = 2T(n/2) + O(n)$

Step 1: identify a, b, d.

- How many recursive calls? $a = 2$
- By what factor does input shrink?

Master Method: Example 1 — Merge Sort

A sanity check on something we already know

Recurrence: $T(n) = 2T(n/2) + O(n)$

Step 1: identify a, b, d.

- How many recursive calls? $a = 2$
- By what factor does input shrink? $b = 2$
- Exponent of outside work? (merge is linear)

Master Method: Example 1 — Merge Sort

A sanity check on something we already know

Recurrence: $T(n) = 2T(n/2) + O(n)$

Step 1: identify a, b, d.

- How many recursive calls? $a = 2$
- By what factor does input shrink? $b = 2$
- Exponent of outside work? (merge is linear) $d = 1$

Master Method: Example 1 — Merge Sort

A sanity check on something we already know

Recurrence: $T(n) = 2T(n/2) + O(n)$

Step 1: identify a, b, d.

- How many recursive calls? $a = 2$
- By what factor does input shrink? $b = 2$
- Exponent of outside work? (merge is linear) $d = 1$

Step 2: compare a to b^d.

$$b^d = 2^1 = 2 \qquad a = 2 \qquad \Rightarrow \quad a = b^d \quad \textbf{Case 1}$$

Master Method: Example 1 — Merge Sort

A sanity check on something we already know

Recurrence: $T(n) = 2T(n/2) + O(n)$

Step 1: identify a, b, d.

- How many recursive calls? $a = 2$
- By what factor does input shrink? $b = 2$
- Exponent of outside work? (merge is linear) $d = 1$

Step 2: compare a to b^d.

$$b^d = 2^1 = 2 \qquad a = 2 \qquad \Rightarrow \quad a = b^d \quad \textbf{Case 1}$$

$$T(n) = O(n^d \log n) = O(n \log n) \quad \checkmark$$

Master Method: Example 2 — Binary Search

Only one recursive call

Recurrence: $T(n) = T(n/2) + O(1)$

Step 1: identify a, b, d.

- How many recursive calls? (recurse on one half, discard other)

Master Method: Example 2 — Binary Search

Only one recursive call

Recurrence: $T(n) = T(n/2) + O(1)$

Step 1: identify a, b, d.

- How many recursive calls? (recurse on one half, discard other) $a = 1$
- By what factor does input shrink?

Master Method: Example 2 — Binary Search

Only one recursive call

Recurrence: $T(n) = T(n/2) + O(1)$

Step 1: identify a, b, d.

- How many recursive calls? (recurse on one half, discard other) $a = 1$
- By what factor does input shrink? $b = 2$
- Exponent of outside work? (one comparison)

Master Method: Example 2 — Binary Search

Only one recursive call

Recurrence: $T(n) = T(n/2) + O(1)$

Step 1: identify a, b, d.

- How many recursive calls? (recurse on one half, discard other) $a = 1$
- By what factor does input shrink? $b = 2$
- Exponent of outside work? (one comparison) $d = 0$

Master Method: Example 2 — Binary Search

Only one recursive call

Recurrence: $T(n) = T(n/2) + O(1)$

Step 1: identify a, b, d.

- How many recursive calls? (recurse on one half, discard other) $a = 1$
- By what factor does input shrink? $b = 2$
- Exponent of outside work? (one comparison) $d = 0$

Step 2: compare a to b^d.

$$b^d = 2^0 = 1 \qquad a = 1 \qquad \Rightarrow \quad a = b^d \quad \textbf{Case 1}$$

Master Method: Example 2 — Binary Search

Only one recursive call

Recurrence: $T(n) = T(n/2) + O(1)$

Step 1: identify a, b, d.

- How many recursive calls? (recurse on one half, discard other) $a = 1$
- By what factor does input shrink? $b = 2$
- Exponent of outside work? (one comparison) $d = 0$

Step 2: compare a to b^d.

$$b^d = 2^0 = 1 \qquad a = 1 \qquad \Rightarrow \quad a = b^d \quad \textbf{Case 1}$$

$$T(n) = O(n^d \log n) = O(\log n) \quad \checkmark$$

Master Method: Example 3 — Naive Integer Multiplication

Four recursive calls — does it beat grade school?

Recurrence: $T(n) = 4T(n/2) + O(n)$ (four sub-multiplications, linear recombination)

Identify a, b, d:

Master Method: Example 3 — Naive Integer Multiplication

Four recursive calls — does it beat grade school?

Recurrence: $T(n) = 4T(n/2) + O(n)$ (four sub-multiplications, linear recombination)

Identify a, b, d: $a = 4, \quad b = 2, \quad d = 1$

Master Method: Example 3 — Naive Integer Multiplication

Four recursive calls — does it beat grade school?

Recurrence: $T(n) = 4T(n/2) + O(n)$ (four sub-multiplications, linear recombination)

Identify a, b, d: $a = 4, \quad b = 2, \quad d = 1$

Compare a to b^d:

$$b^d = 2^1 = 2 \qquad a = 4 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

Master Method: Example 3 — Naive Integer Multiplication

Four recursive calls — does it beat grade school?

Recurrence: $T(n) = 4T(n/2) + O(n)$ (four sub-multiplications, linear recombination)

Identify a, b, d: $a = 4, \quad b = 2, \quad d = 1$

Compare a to b^d:

$$b^d = 2^1 = 2 \qquad a = 4 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

Result:

$$T(n) = O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)$$

Master Method: Example 3 — Naive Integer Multiplication

Four recursive calls — does it beat grade school?

Recurrence: $T(n) = 4T(n/2) + O(n)$ (four sub-multiplications, linear recombination)

Identify a, b, d: $a = 4, \quad b = 2, \quad d = 1$

Compare a to b^d:

$$b^d = 2^1 = 2 \qquad a = 4 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

Result:

$$T(n) = O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)$$

Both are $\Theta(n^2)$ — no improvement over grade school. But what if we reduce to 3 calls?

Master Method: Example 4 — Karatsuba

Three calls instead of four — does it matter?

Recurrence: $T(n) = 3T(n/2) + O(n)$

Identify a, b, d:

Master Method: Example 4 — Karatsuba

Three calls instead of four — does it matter?

Recurrence: $T(n) = 3T(n/2) + O(n)$

Identify a, b, d: $a = 3, \quad b = 2, \quad d = 1$

Master Method: Example 4 — Karatsuba

Three calls instead of four — does it matter?

Recurrence: $T(n) = 3T(n/2) + O(n)$

Identify a, b, d: $a = 3, \quad b = 2, \quad d = 1$

Compare a to b^d:

$$b^d = 2^1 = 2 \qquad a = 3 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

Master Method: Example 4 — Karatsuba

Three calls instead of four — does it matter?

Recurrence: $T(n) = 3T(n/2) + O(n)$

Identify a, b, d: $a = 3, \quad b = 2, \quad d = 1$

Compare a to b^d:

$$b^d = 2^1 = 2 \qquad a = 3 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

$$T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

Master Method: Example 4 — Karatsuba

Three calls instead of four — does it matter?

Recurrence: $T(n) = 3T(n/2) + O(n)$

Identify a, b, d: $a = 3, \quad b = 2, \quad d = 1$

Compare a to b^d:

$$b^d = 2^1 = 2 \qquad a = 3 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

$$T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

Compare to naive recursive ($a = 4$): $O(n^{\log_2 4}) = O(n^2)$. Dropping one recursive call changed the exponent from 2 to 1.585 — the Gauss trick pays off.

Master Method: Example 5 — Strassen Matrix Multiplication

The same idea applied to matrix multiplication

Recurrence: $T(n) = 7T(n/2) + O(n^2)$ [7 sub-multiplications; $O(n^2)$ to add matrices]

Identify a, b, d:

Master Method: Example 5 — Strassen Matrix Multiplication

The same idea applied to matrix multiplication

Recurrence: $T(n) = 7T(n/2) + O(n^2)$ [7 sub-multiplications; $O(n^2)$ to add matrices]

Identify a, b, d: $a = 7, \quad b = 2, \quad d = 2$

Master Method: Example 5 — Strassen Matrix Multiplication

The same idea applied to matrix multiplication

Recurrence: $T(n) = 7T(n/2) + O(n^2)$ [7 sub-multiplications; $O(n^2)$ to add matrices]

Identify a, b, d: $a = 7, \quad b = 2, \quad d = 2$

Compare a to b^d:

$$b^d = 2^2 = 4 \qquad a = 7 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

Master Method: Example 5 — Strassen Matrix Multiplication

The same idea applied to matrix multiplication

Recurrence: $T(n) = 7T(n/2) + O(n^2)$ [7 sub-multiplications; $O(n^2)$ to add matrices]

Identify a, b, d: $a = 7, \quad b = 2, \quad d = 2$

Compare a to b^d:

$$b^d = 2^2 = 4 \qquad a = 7 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

Master Method: Example 5 — Strassen Matrix Multiplication

The same idea applied to matrix multiplication

Recurrence: $T(n) = 7T(n/2) + O(n^2)$ [7 sub-multiplications; $O(n^2)$ to add matrices]

Identify a, b, d: $a = 7, \quad b = 2, \quad d = 2$

Compare a to b^d:

$$b^d = 2^2 = 4 \qquad a = 7 \qquad \Rightarrow \quad a > b^d \quad \textbf{Case 3}$$

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

Naive matrix multiply is $O(n^3)$. Strassen saves one recursive call ($8 \to 7$) — same trick as Karatsuba, one level up.

Master Method: Example 6 — Triggering Case 2

Every example so far has been Case 1 or Case 3 — let's fix that

Recurrence: $T(n) = 2T(n/2) + O(n^2)$ [merge sort but with a quadratic combine]

Identify a, b, d:

Master Method: Example 6 — Triggering Case 2

Every example so far has been Case 1 or Case 3 — let's fix that

Recurrence: $T(n) = 2T(n/2) + O(n^2)$ [merge sort but with a quadratic combine]

Identify a, b, d: $a = 2, \quad b = 2, \quad d = 2$

Master Method: Example 6 — Triggering Case 2

Every example so far has been Case 1 or Case 3 — let's fix that

Recurrence: $T(n) = 2T(n/2) + O(n^2)$ [merge sort but with a quadratic combine]

Identify a, b, d: $a = 2, \quad b = 2, \quad d = 2$

Compare a to b^d:

$$b^d = 2^2 = 4 \qquad a = 2 \qquad \Rightarrow \quad a < b^d \quad \textbf{Case 2}$$

Master Method: Example 6 — Triggering Case 2

Every example so far has been Case 1 or Case 3 — let's fix that

Recurrence: $T(n) = 2T(n/2) + O(n^2)$ [merge sort but with a quadratic combine]

Identify a, b, d: $a = 2, \quad b = 2, \quad d = 2$

Compare a to b^d:

$$b^d = 2^2 = 4 \qquad a = 2 \qquad \Rightarrow \quad a < b^d \quad \textbf{Case 2}$$

$$T(n) = O(n^d) = O(n^2)$$

Master Method: Example 6 — Triggering Case 2

Every example so far has been Case 1 or Case 3 — let's fix that

Recurrence: $T(n) = 2T(n/2) + O(n^2)$ [merge sort but with a quadratic combine]

Identify a, b, d: $a = 2, \quad b = 2, \quad d = 2$

Compare a to b^d:

$$b^d = 2^2 = 4 \qquad a = 2 \qquad \Rightarrow \quad a < b^d \quad \textbf{Case 2}$$

$$T(n) = O(n^d) = O(n^2)$$

The combine step dominates so completely that the recursion is irrelevant — the whole algorithm costs no more than the single outermost call.

When the Master Method Doesn't Apply

The master method requires $T(n) = aT(n/b) + O(n^d)$ with a, b, d constant.

It fails when:

- **Unequal subproblems:** $T(n) = T(n/3) + T(2n/3) + O(n)$ $\rightarrow$ recursion tree
- **Subtraction:** $T(n) = T(n-1) + O(n)$ $\rightarrow$ unrolling
- a **depends on** n**:** $T(n) = nT(n/2) + O(1)$ $\rightarrow$ recursion tree
- **Non-polynomial combine:** $T(n) = 2T(n/2) + O(n \log n)$ $\rightarrow$ extended MT

The good news: virtually every standard divide-and-conquer algorithm you will encounter fits the format. The exceptions are the interesting ones.