Lecture 7: Recursion Applications

Multi-Call Recursion, Divide & Conquer, and Binary Search

Comp 111 — Programming 2

Forman Christian University

The Phone Book Challenge

Find a Name!

"I'm thinking of a word in a 1000-page dictionary.
Find **PYTHON**. *How many guesses?"*

Dictionary

1000 pages

Find a Name!

"I'm thinking of a word in a 1000-page dictionary.
Find **PYTHON**. *How many guesses?"*

Random flipping?
Up to 1000 tries!

Dictionary

1000 pages

Find a Name!

*"I'm thinking of a word in a 1000-page dictionary.
Find* **PYTHON**. *How many guesses?"*

Random flipping?
Up to 1000 tries!

Dictionary
1000 pages

Open to middle?
Much smarter!

The Power of Halving

Items	One-by-one	Halving
1,000	1,000 checks	**10 checks**
1,000,000	1,000,000 checks	**20 checks**
1 billion	1 billion checks	**30 checks**

The Power of Halving

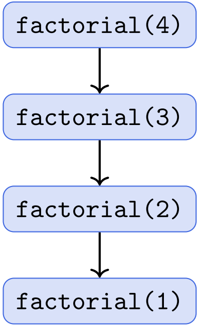
Items	One-by-one	Halving
1,000	1,000 checks	**10 checks**
1,000,000	1,000,000 checks	**20 checks**
1 billion	1 billion checks	**30 checks**

Google searches 30+ billion pages in ~35 comparisons!

Multi-Call Recursion

Linear vs Tree Recursion

Linear (One Call)



factorial(4)

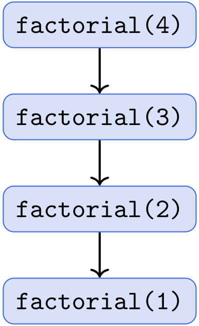
factorial(3)

factorial(2)

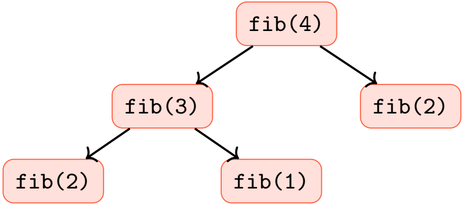
factorial(1)

Linear vs Tree Recursion

Linear (One Call)



Tree (Multiple Calls)



What if a function calls itself **twice**?

The Fibonacci Sequence

Found in nature: Sunflowers, shells, art, music!

Value:	0	1	1	2	3	5	8	13	· · ·
Position:	0	1	2	3	4	5	6	7	

The Fibonacci Sequence

Found in nature: Sunflowers, shells, art, music!

Value:	0	1	1	2	3	5	8	13	⋯
Position:	0	1	2	3	4	5	6	7	

Each number = **sum of the two before it**

$$fib(n) = fib(n-1) + fib(n-2)$$

Coding Fibonacci

```python
def fib(n):
    # Base cases
    if n == 0:
        return 0
    if n == 1:
        return 1

    # Two recursive calls!
    return fib(n-1) + fib(n-2)

print(fib(6))    # 8
print(fib(10))   # 55
```

Coding Fibonacci

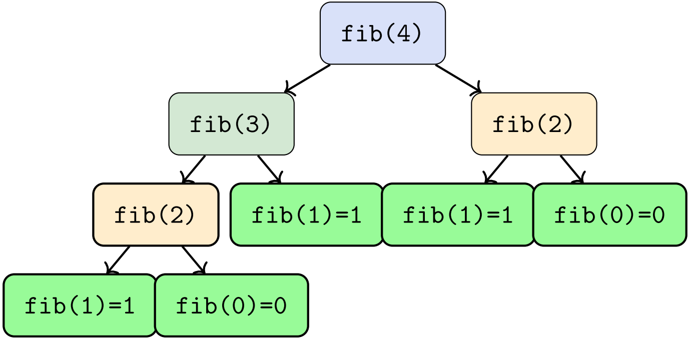
```python
def fib(n):
    # Base cases
    if n == 0:
        return 0
    if n == 1:
        return 1

    # Two recursive calls!
    return fib(n-1) + fib(n-2)

print(fib(6))    # 8
print(fib(10))   # 55
```

$fib(0) = 0$

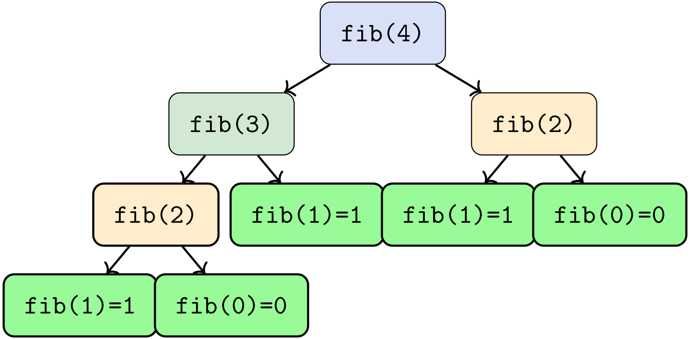
$fib(1) = 1$

$fib(n) = fib(n-1) + fib(n-2)$

Tracing fib(4)



Tracing fib(4)



Notice: fib(2) is calculated **twice**! Wasteful!

The Efficiency Problem

```python
call_count = 0

def fib_counted(n):
    global call_count
    call_count += 1
    if n <= 1:
        return n
    return fib_counted(n-1) + \
           fib_counted(n-2)

fib_counted(20)
print(f"Calls: {call_count}")
# 21,891 calls!
```

n	Calls	Time
10	177	instant
20	21,891	instant
30	2.7M	~1 sec
35	29.9M	~10 sec
40	331M	~2 min
50	40B	~6 hrs

The Efficiency Problem

```python
call_count = 0

def fib_counted(n):
    global call_count
    call_count += 1
    if n <= 1:
        return n
    return fib_counted(n-1) + \
           fib_counted(n-2)

fib_counted(20)
print(f"Calls: {call_count}")
# 21,891 calls!
```

n	Calls	Time
10	177	instant
20	21,891	instant
30	2.7M	~1 sec
35	29.9M	~10 sec
40	331M	~2 min
50	40B	~6 hrs

Week 5: We'll fix this with "memoization"!

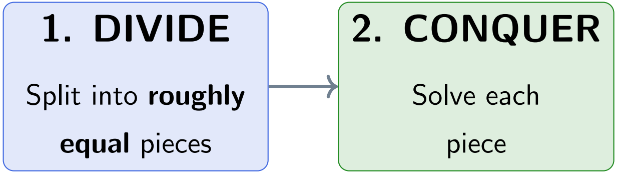
Divide & Conquer

The Pizza Strategy

1. DIVIDE

Split into **roughly equal** pieces

The Pizza Strategy



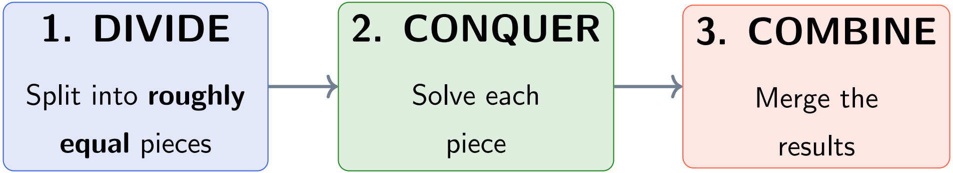
1. DIVIDE

Split into **roughly equal** pieces

2. CONQUER

Solve each piece

The Pizza Strategy



1. DIVIDE

Split into **roughly equal** pieces

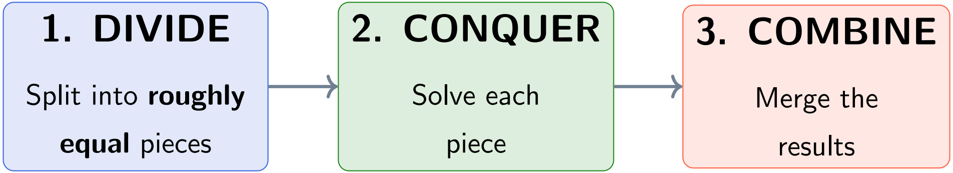
2. CONQUER

Solve each piece

3. COMBINE

Merge the results

The Pizza Strategy



1. DIVIDE

Split into **roughly equal** pieces

2. CONQUER

Solve each piece

3. COMBINE

Merge the results

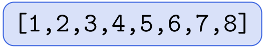
"How do you eat a large pizza?
Slice it, eat each slice, digest!"

D&C Template

```python
def divide_and_conquer(problem):
    # Base case:  small enough to solve directly
    if is_simple(problem):
        return simple_solution(problem)

    # DIVIDE: split into smaller parts
    left, right = split(problem)

    # CONQUER: solve each part
    left_result = divide_and_conquer(left)
    right_result = divide_and_conquer(right)

    # COMBINE: merge results
    return combine(left_result, right_result)
```

Example: Sum an Array

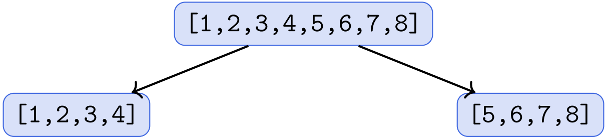
Sum $[1, 2, 3, 4, 5, 6, 7, 8]$ by splitting in half:



`[1,2,3,4,5,6,7,8]`

Example: Sum an Array

Sum $[1, 2, 3, 4, 5, 6, 7, 8]$ by splitting in half:



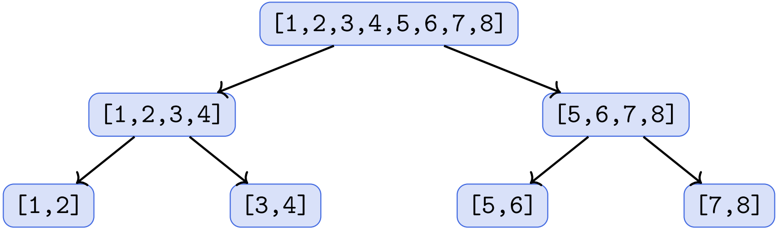
[1,2,3,4,5,6,7,8]

[1,2,3,4]

[5,6,7,8]

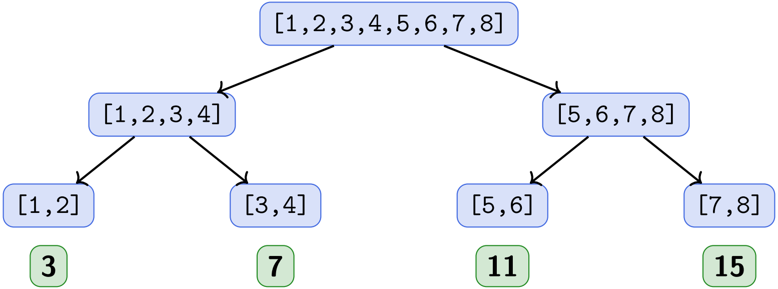
Example: Sum an Array

Sum $[1, 2, 3, 4, 5, 6, 7, 8]$ by splitting in half:



Example: Sum an Array

Sum $[1, 2, 3, 4, 5, 6, 7, 8]$ by splitting in half:



[1,2,3,4,5,6,7,8]

[1,2,3,4]

[5,6,7,8]

[1,2]

[3,4]

[5,6]

[7,8]

3

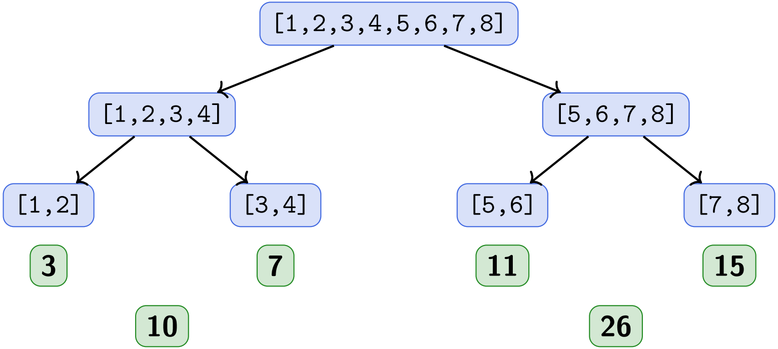
7

11

15

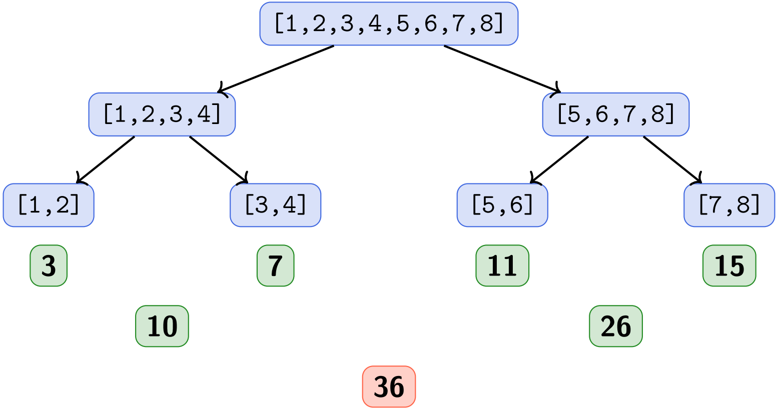
Example: Sum an Array

Sum $[1, 2, 3, 4, 5, 6, 7, 8]$ by splitting in half:



Example: Sum an Array

Sum $[1, 2, 3, 4, 5, 6, 7, 8]$ by splitting in half:



[1,2,3,4,5,6,7,8]

[1,2,3,4]

[5,6,7,8]

[1,2]

[3,4]

[5,6]

[7,8]

3

7

11

15

10

26

36

Sum with D&C Code

```python
def sum_dc(numbers):
    # Base cases
    if len(numbers) == 0:
        return 0
    if len(numbers) == 1:
        return numbers[0]

    # DIVIDE
    mid = len(numbers) // 2
    left = numbers[:mid]
    right = numbers[mid:]

    # CONQUER + COMBINE
    return sum_dc(left) + sum_dc(right)

print(sum_dc([1,2,3,4,5,6,7,8]))  # 36
```

Why Split in Half?

Size	One at a time	Halving
8	8 levels	**3 levels**
1,000	1,000 levels	**10 levels**
1,000,000	1,000,000 levels	**20 levels**

Why Split in Half?

Size	**One at a time**	**Halving**
8	8 levels	**3 levels**
1,000	1,000 levels	**10 levels**
1,000,000	1,000,000 levels	**20 levels**

Every halving adds just ONE level!

This is **logarithmic growth** — incredibly efficient!

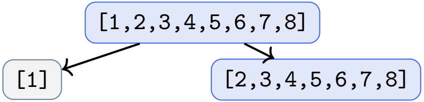
What If We Don’t Split Equally?

Same problem — but split as `[first]` + `[rest]` instead of halving:

```
[1,2,3,4,5,6,7,8]
```

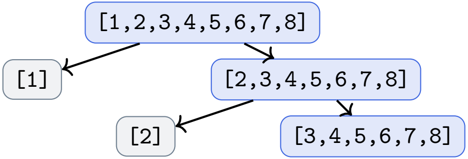
What If We Don’t Split Equally?

Same problem — but split as `[first]` + `[rest]` instead of halving:



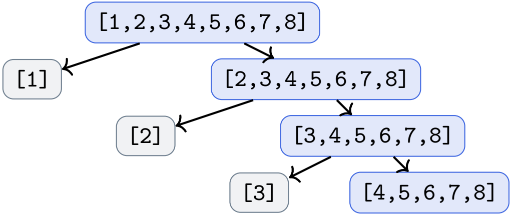
What If We Don't Split Equally?

Same problem — but split as `[first]` + `[rest]` instead of halving:



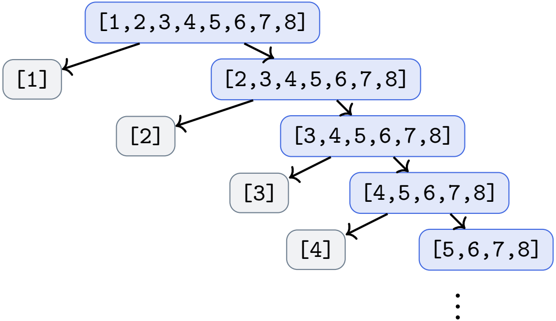
What If We Don’t Split Equally?

Same problem — but split as `[first]` + `[rest]` instead of halving:



What If We Don't Split Equally?

Same problem — but split as `[first]` + `[rest]` instead of halving:



[1,2,3,4,5,6,7,8]

[1]

[2,3,4,5,6,7,8]

[2]

[3,4,5,6,7,8]

[3]

[4,5,6,7,8]

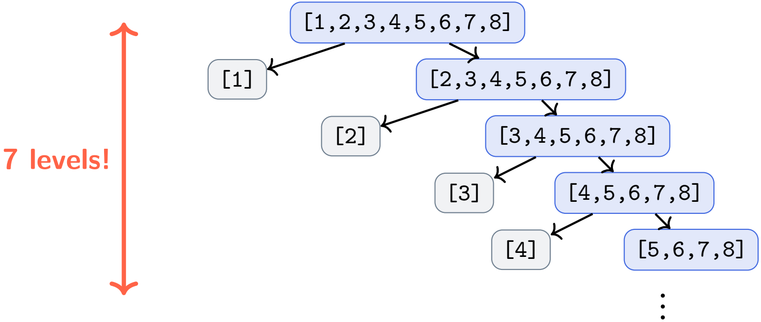
[4]

[5,6,7,8]

⋮

What If We Don't Split Equally?

Same problem — but split as `[first]` + `[rest]` instead of halving:



Halving: 3 levels vs **One-at-a-time: 7 levels**

Unequal splits ⇒ deeper tree ⇒ more work!

How We Talk About Speed: Big-O

Why Compare Algorithms?

We just saw that **halving** is much better than **one-at-a-time**.

Why Compare Algorithms?

We just saw that **halving** is much better than **one-at-a-time**.

But how do we say that *precisely*?

- "My laptop finds it faster" — laptops differ
- "It takes 0.002 seconds" — hardware-dependent

Why Compare Algorithms?

We just saw that **halving** is much better than **one-at-a-time**.

But how do we say that *precisely*?

- "My laptop finds it faster" — laptops differ
- "It takes 0.002 seconds" — hardware-dependent

We need a language that describes scaling

— how work grows as input grows.

Thinking About Growth

Your algorithm takes **10 steps** for $n = 10$. When n doubles to **20**, what happens?

Thinking About Growth

Your algorithm takes **10 steps** for $n = 10$. When n doubles to **20**, what happens?

- Still **10 steps** Constant — doesn't depend on size

Thinking About Growth

Your algorithm takes **10 steps** for $n = 10$. When n doubles to **20**, what happens?

- Still **10 steps** Constant — doesn't depend on size
- **11 steps** Logarithmic — barely grew!

Thinking About Growth

Your algorithm takes **10 steps** for $n = 10$. When n doubles to **20**, what happens?

• Still **10 steps**	Constant — doesn't depend on size
• **11 steps**	Logarithmic — barely grew!
• **20 steps**	Linear — doubled with input

Thinking About Growth

Your algorithm takes **10 steps** for $n = 10$. When n doubles to **20**, what happens?

- Still **10 steps** Constant — doesn't depend on size
- **11 steps** Logarithmic — barely grew!
- **20 steps** Linear — doubled with input
- **400 steps** Quadratic — exploded!

Thinking About Growth

Your algorithm takes **10 steps** for $n = 10$. When n doubles to **20**, what happens?

Still **10 steps**	Constant — doesn't depend on size
11 steps	Logarithmic — barely grew!
20 steps	Linear — doubled with input
400 steps	Quadratic — exploded!

The pattern of growth is what matters — not the exact count.

Big-O: The Shorthand

Computer scientists give these growth patterns a name. The letter *n* means "size of input":

- **Constant** ⟶ **O(1)** *Same work, no matter what*

Big-O: The Shorthand

Computer scientists give these growth patterns a name. The letter n means “size of input”:

- **Constant** ⟶ **O(1)** *Same work, no matter what*
- **Logarithmic** ⟶ **O(log n)** *Phone-book halving: 1000 pages → 10 steps*

Big-O: The Shorthand

Computer scientists give these growth patterns a name. The letter *n* means "size of input":

- **Constant** ⟶ **O(1)** *Same work, no matter what*
- **Logarithmic** ⟶ **O(log n)** *Phone-book halving: 1000 pages → 10 steps*
- **Linear** ⟶ **O(n)** *Read every page one by one*

Big-O: The Shorthand

Computer scientists give these growth patterns a name. The letter *n* means "size of input":

- **Constant** ⟶ **O(1)** *Same work, no matter what*
- **Logarithmic** ⟶ **O(log n)** *Phone-book halving: 1000 pages → 10 steps*
- **Linear** ⟶ **O(n)** *Read every page one by one*
- **Quadratic** ⟶ **O(n^2)** *For every page, re-read all pages*

Big-O: The Shorthand

Computer scientists give these growth patterns a name. The letter *n* means "size of input":

- **Constant** ⟶ **O(1)** *Same work, no matter what*
- **Logarithmic** ⟶ **O(log n)** *Phone-book halving: 1000 pages → 10 steps*
- **Linear** ⟶ **O(n)** *Read every page one by one*
- **Quadratic** ⟶ **O(n^2)** *For every page, re-read all pages*

Big-O describes the **shape** of growth — not the exact time.

Big-O: The Doubling Question

"If I double the input size, how much more work?"

Big-O	Name	Doubling input means. . .
$O(1)$	Constant	Same amount of work

Big-O: The Doubling Question

"If I double the input size, how much more work?"

Big-O	Name	Doubling input means. . .
O(1)	Constant	Same amount of work
O(log n)	Logarithmic	**ONE** more step

Big-O: The Doubling Question

"If I double the input size, how much more work?"

Big-O	Name	Doubling input means. . .
O(1)	Constant	Same amount of work
O(log n)	Logarithmic	**ONE** more step
O(n)	Linear	Double the work

Big-O: The Doubling Question

"If I double the input size, how much more work?"

Big-O	Name	Doubling input means...
O(1)	Constant	Same amount of work
O(log n)	Logarithmic	**ONE** more step
O(n)	Linear	Double the work
O(n log n)	Linearithmic	Slightly more than double

Big-O: The Doubling Question

"If I double the input size, how much more work?"

Big-O	Name	Doubling input means...
$O(1)$	Constant	Same amount of work
$O(\log n)$	Logarithmic	**ONE** more step
$O(n)$	Linear	Double the work
$O(n \log n)$	Linearithmic	Slightly more than double
$O(n^2)$	Quadratic	**FOUR** times the work

Big-O: Concrete Numbers

n	O(log n)	O(n)	O(n log n)	O(n^2)
10	3	10	33	100
1,000	10	1,000	10,000	1,000,000
1,000,000	**20**	1,000,000	20,000,000	$\mathbf{10^{12}}$

Big-O: Concrete Numbers

n	O(log n)	O(n)	O(n log n)	$O(n^2)$
10	3	10	33	100
1,000	10	1,000	10,000	1,000,000
1,000,000	**20**	1,000,000	20,000,000	**10^{12}**

O(n) vs $O(n^2)$ at $n = 10^6$: the difference between **one second** and **12 days**. Algorithms matter!

You Try: Name That Growth

What growth pattern does each snippet show?
Choose: **Constant**, **Logarithmic**, **Linear**, or **Quadratic**

```python
# A
return arr[0]

# B
for x in arr:
    print(x)

# C
for i in arr:
    for j in arr:
        print(i, j)

# D  -- binary search halving
```

You Try: Name That Growth

What growth pattern does each snippet show?
Choose: **Constant**, **Logarithmic**, **Linear**, or **Quadratic**

```
# A
return arr[0]

# B
for x in arr:
    print(x)

# C
for i in arr:
    for j in arr:
        print(i, j)

# D  -- binary search halving
```

A: $O(1)$ **B:** $O(n)$ **C:** $O(n^2)$ **D:** $O(\log n)$

You Try: Pick the Winner

1. An app works fine for 100 users but **freezes** for 10,000. Quadrupling the users makes it **sixteen** times slower.

2. Two search algorithms for $n = 1{,}000{,}000$:
   Algorithm A: **O(n)** vs Algorithm B: **O(log n)**

You Try: Pick the Winner

1. An app works fine for 100 users but **freezes** for 10,000. Quadrupling the users makes it **sixteen** times slower.
   What Big-O class? → $O(n^2)$

2. Two search algorithms for $n = 1{,}000{,}000$:
   Algorithm A: **O(n)** vs Algorithm B: **O(log n)**

You Try: Pick the Winner

1. An app works fine for 100 users but **freezes** for 10,000. Quadrupling the users makes it **sixteen** times slower.
   What Big-O class? → O(n^2)

2. Two search algorithms for $n = 1{,}000{,}000$:
   Algorithm A: **O(n)** vs Algorithm B: **O(log n)**
   How many more steps does A take?

You Try: Pick the Winner

1. An app works fine for 100 users but **freezes** for 10,000. Quadrupling the users makes it **sixteen** times slower.
   What Big-O class? → O(n^2)

2. Two search algorithms for $n = 1,000,000$:
   Algorithm A: **O(n)** vs Algorithm B: **O(log n)**
   How many more steps does A take?
   A: **1,000,000** steps B: **20** steps A is **50,000**× slower

You Try: Pick the Winner

1. An app works fine for 100 users but **freezes** for 10,000. Quadrupling the users makes it **sixteen** times slower.
   What Big-O class? → $O(n^2)$

2. Two search algorithms for $n = 1{,}000{,}000$:
   Algorithm A: **O(n)** vs Algorithm B: **O(log n)**
   How many more steps does A take?
   A: **1,000,000** steps B: **20** steps A is **50,000**× slower

Choosing the right Big-O class is one of the most impactful decisions in software.

Binary Search

The Guessing Game

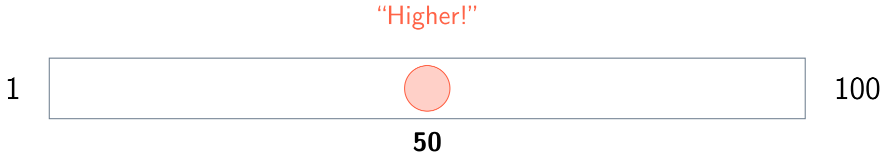
"I'm thinking of a number between 1 and 100."

1

100

The Guessing Game

“I’m thinking of a number between 1 and 100.”



“Higher!”

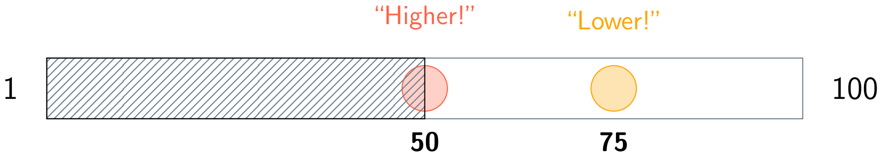
1

100

50

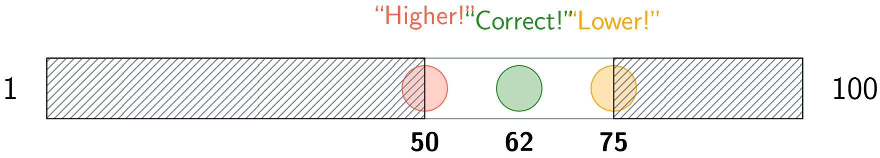
The Guessing Game

"I'm thinking of a number between 1 and 100."



The Guessing Game

"I'm thinking of a number between 1 and 100."



3 guesses to find a number among 100!

Binary Search Strategy

1. Look at the **MIDDLE** element

Binary Search Strategy

1. Look at the **MIDDLE** element

2. If it's the target: Done!

Binary Search Strategy

1. Look at the **MIDDLE** element

2. If it's the target: Done!

3. If target is SMALLER: search LEFT half

Binary Search Strategy

1. Look at the **MIDDLE** element

2. If it's the target: Done!

3. If target is SMALLER: search LEFT half

4. If target is LARGER: search RIGHT half

Binary Search Strategy

1. Look at the **MIDDLE** element

2. If it's the target: Done!

3. If target is SMALLER: search LEFT half

4. If target is LARGER: search RIGHT half

5. Repeat until found (or not there)

Binary Search Strategy

1. Look at the **MIDDLE** element

2. If it's the target: Done!

3. If target is SMALLER: search LEFT half

4. If target is LARGER: search RIGHT half

5. Repeat until found (or not there)

Requirement: Data must be **SORTED**!

Visual Walkthrough

Find **23** in this sorted list:

2	5	8	12	16	23	38	56	72	91
0	1	2	3	4	5	6	7	8	9

Visual Walkthrough

Find **23** in this sorted list:

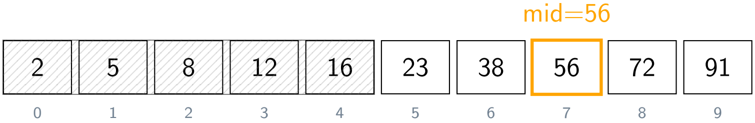
mid=16

2	5	8	12	16	23	38	56	72	91
0	1	2	3	4	5	6	7	8	9

23 > 16, search **RIGHT**

Visual Walkthrough

Find **23** in this sorted list:



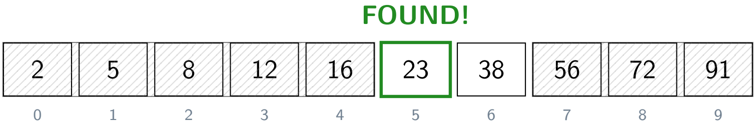
mid=56

2	5	8	12	16	23	38	56	72	91
0	1	2	3	4	5	6	7	8	9

23 < 56, search **LEFT**

Visual Walkthrough

Find **23** in this sorted list:



FOUND!

2	5	8	12	16	23	38	56	72	91
0	1	2	3	4	5	6	7	8	9

23 == 23 at index 5!

Only **3 comparisons** for 10 elements!

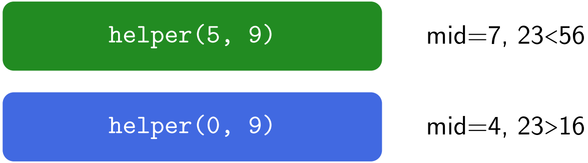
Binary Search Code

```python
def binary_search(arr, target):
    """Public function: clean interface."""
    def helper(low, high):
        if low > high:              # Base case: not found
            return -1
        mid = (low + high) // 2     # DIVIDE: find middle
        if arr[mid] == target:      # Check middle
            return mid              # Found!
        elif target < arr[mid]:
            return helper(low, mid - 1)
        else:
            return helper(mid + 1, high)
    return helper(0, len(arr) - 1)

nums = [2,5,8,12,16,23,38,56,72,91]
print(binary_search(nums, 23))      # 5
```

Binary Search: Call Stack

`helper(0, 9)` mid=4, 23>16

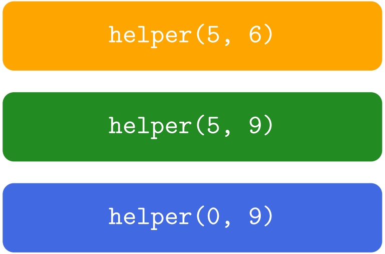
Binary Search: Call Stack



`helper(5, 9)` mid=7, 23<56

`helper(0, 9)` mid=4, 23>16

Binary Search: Call Stack

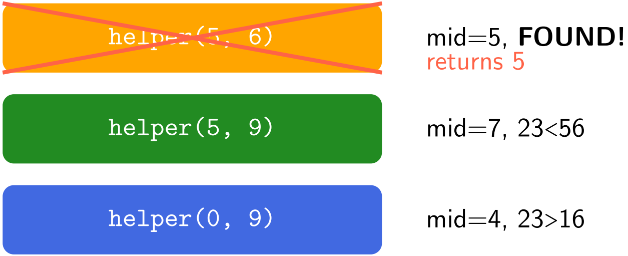


`helper(5, 6)` mid=5, **FOUND!**

`helper(5, 9)` mid=7, 23<56

`helper(0, 9)` mid=4, 23>16

Binary Search: Call Stack

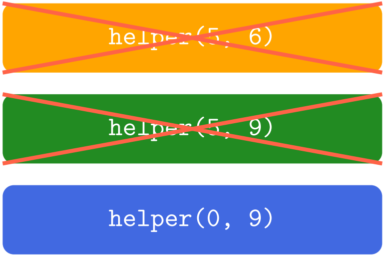


~~helper(5, 6)~~ mid=5, **FOUND!** returns 5

helper(5, 9) mid=7, 23<56

helper(0, 9) mid=4, 23>16

Binary Search: Call Stack



~~`helper(5, 6)`~~

mid=5, **FOUND!**
returns 5

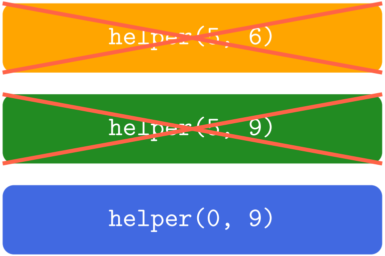
~~`helper(5, 9)`~~

mid=7, 23<56
returns 5

`helper(0, 9)`

mid=4, 23>16

Binary Search: Call Stack



~~`helper(5, 6)`~~

mid=5, **FOUND!**
returns 5

~~`helper(5, 9)`~~

mid=7, 23<56
returns 5

`helper(0, 9)`

mid=4, 23>16
returns 5

Result: index 5

Helper captures `arr` *and* `target` *from outer function — only needs* `low` *and* `high`*!*

Why O(log n)?

Each step eliminates **HALF** the remaining items:

$$\left.\begin{array}{c}\text{Start: } n \text{ items}\\ \text{Step 1: } n/2 \text{ items}\\ \text{Step 2: } n/4 \text{ items}\\ \text{Step 3: } n/8 \text{ items}\\ \vdots\\ \text{Step } k\text{: } n/2^k = 1 \text{ item}\end{array}\right\} \begin{array}{l} k = \log_2(n)\\ \text{steps!}\end{array}$$

Why O(log n)?

Each step eliminates **HALF** the remaining items:

$$\left.\begin{array}{c}\text{Start: } n \text{ items}\\ \text{Step 1: } n/2 \text{ items}\\ \text{Step 2: } n/4 \text{ items}\\ \text{Step 3: } n/8 \text{ items}\\ \vdots\\ \text{Step } k\text{: } n/2^k = 1 \text{ item}\end{array}\right\} \begin{array}{l} k = \log_2(n)\\ \text{steps!}\end{array}$$

8 billion people? Only ~33 comparisons!

Linear vs Binary Search

Items (n)	Linear O(n)	Binary O(log n)
100	100	**7**
10,000	10,000	**14**
1,000,000	1,000,000	**20**
1 billion	1,000,000,000	**30**

Linear vs Binary Search

Items (n)	**Linear O(n)**	**Binary O(log n)**
100	100	**7**
10,000	10,000	**14**
1,000,000	1,000,000	**20**
1 billion	1,000,000,000	**30**

This is why Google feels instant!

Practice Time

You Try: Trace Binary Search

Find **15** in: $[2, 5, 8, 12, 16, 23, 38]$

2	5	8	12	16	23	38
0	1	2	3	4	5	6

Questions:

1. What's the first middle element checked?
2. Which half do we search next?
3. What does the function return?

Solution: Finding 15

Step 1: low=0, high=6, mid=3

arr[3]=12, 15>12 → search RIGHT

Solution: Finding 15

Step 1: low=0, high=6, mid=3

arr[3]=12, 15>12 → search RIGHT

Step 2: low=4, high=6, mid=5

arr[5]=23, 15<23 → search LEFT

Solution: Finding 15

Step 1: low=0, high=6, mid=3

arr[3]=12, $15>12$ → search RIGHT

Step 2: low=4, high=6, mid=5

arr[5]=23, $15<23$ → search LEFT

Step 3: low=4, high=4, mid=4

arr[4]=16, $15<16$ → search LEFT

Solution: Finding 15

Step 1: low=0, high=6, mid=3

arr[3]=12, 15>12 → search RIGHT

Step 2: low=4, high=6, mid=5

arr[5]=23, 15<23 → search LEFT

Step 3: low=4, high=4, mid=4

arr[4]=16, 15<16 → search LEFT

Step 4: low=4, high=3

low > high → **NOT FOUND, return -1**

Challenge: Find First Occurrence

What if there are **duplicates**?

```
arr = [1, 2, 2, 2, 2, 3, 4, 5]
target = 2
```

- Standard binary search might return index 3
- We want the **FIRST** occurrence: index **1**

Challenge: Find First Occurrence

What if there are **duplicates**?

```
arr = [1, 2, 2, 2, 2, 3, 4, 5]
target = 2
```

- Standard binary search might return index 3
- We want the **FIRST** occurrence: index **1**

Hint: When you find target, check if it's the first one.

If not, keep searching LEFT!

Summary

Today's Key Takeaways

- **Multi-call recursion** creates a tree of calls
  - Powerful but can be inefficient — we'll fix this in Week 5!

Today's Key Takeaways

- **Multi-call recursion** creates a tree of calls
  - Powerful but can be inefficient — we'll fix this in Week 5!
- **Divide & Conquer** has three steps:
  - DIVIDE → CONQUER → COMBINE

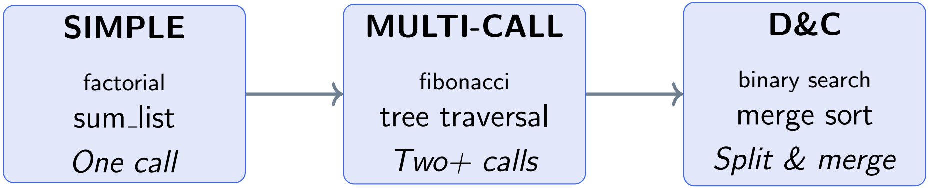
Today's Key Takeaways

- **Multi-call recursion** creates a tree of calls
  - Powerful but can be inefficient — we'll fix this in Week 5!
- **Divide & Conquer** has three steps:
  - DIVIDE → CONQUER → COMBINE
- **Big-O notation** describes how work grows:
  - O(log n) — halving — incredibly fast
  - O(n) — one-by-one — fine for moderate data
  - $O(n^2)$ — nested loops — slow for large data

Today's Key Takeaways

- **Multi-call recursion** creates a tree of calls
  - Powerful but can be inefficient — we'll fix this in Week 5!
- **Divide & Conquer** has three steps:
  - DIVIDE → CONQUER → COMBINE
- **Big-O notation** describes how work grows:
  - O(log n) — halving — incredibly fast
  - O(n) — one-by-one — fine for moderate data
  - O(n^2) — nested loops — slow for large data
- **Binary Search** is D&C for finding items:
  - Only works on **SORTED** data
  - O(log n): 33 steps for 8 billion items!

The Recursion Spectrum



SIMPLE

factorial

sum_list

One call

MULTI-CALL

fibonacci

tree traversal

Two+ calls

D&C

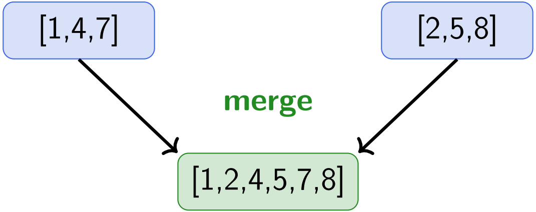
binary search

merge sort

Split & merge

Next Lecture: Merge Sort

"If you have two SORTED piles of cards,
can you combine them into ONE sorted pile?"



That's the key insight behind merge sort!

Homework

Part 1: Multi-Call Practice

- Write `tribonacci(n)` — sum of THREE preceding numbers
- Write `pascal(row, col)` — each value = sum of two above it
- Calculate `trib(10)` and `pascal(4, 2)`

Part 2: Binary Search

- Write `find_last(arr, target)` for duplicates
- Write `int_sqrt(n)` using binary search

Part 3: Reflection

- When would you use linear search instead?

Questions?