Lecture 8: Recursion Applications

Merge Sort & Divide-and-Conquer on Recursive Structures

Comp 111

Forman Christian University

Measuring Time in Python

How Long Does Code Take?

How Long Does Code Take?

Python's `time` module gives us a stopwatch:

```
1  import time
2
3  start = time.time()      # start the stopwatch
4
5  # ... code you want to measure ...
6
7  elapsed = time.time() - start  # stop
8  print(f"Took {elapsed:.2f} seconds")
```

How Long Does Code Take?

Python's `time` module gives us a stopwatch:

```
1  import time
2
3  start = time.time()      # start the stopwatch
4
5  # ... code you want to measure ...
6
7  elapsed = time.time() - start  # stop
8  print(f"Took {elapsed:.2f} seconds")
```

`time.time()` returns the current time in seconds (as a float).

Subtract **start** from **end** ⇒ elapsed time.

Example: Timing a Loop

```
1  import time
2
3  start = time.time()              # note the start time
4
5  total = 0
6  for i in range(10_000_000):
7      total += i
8
9  elapsed = time.time() - start  # note the end time and compute elapsed
10 print(f"Sum = {total}")
11 print(f"Time: {elapsed:.3f} seconds")
```

Example: Timing a Loop

```
1  import time
2
3  start = time.time()              # note the start time
4
5  total = 0
6  for i in range(10_000_000):
7      total += i
8
9  elapsed = time.time() - start  # note the end time and compute elapsed
10 print(f"Sum = {total}")
11 print(f"Time: {elapsed:.3f} seconds")
```

Output:

```
Sum = 49999995000000
Time: 0.573 seconds
```

Comparing Two Algorithms

```
1  import time
2
3  n = 10_000_000
4
5  # Algorithm 1: loop
6  start = time.time()
7  total = 0
8  for i in range(n):
9      total += i
10 print(f"Loop:    {time.time() - start:.3f}s")
11 #-----------------------------------------
12
13 # Algorithm 2: math formula
14 start = time.time()
15 total = n * (n - 1) // 2
16 print(f"Formula: {time.time() - start:.6f}s")
```

Comparing Two Algorithms

```python
import time

n = 10_000_000

# Algorithm 1: loop
start = time.time()
total = 0
for i in range(n):
    total += i
print(f"Loop:    {time.time() - start:.3f}s")
#-----------------------------------------

# Algorithm 2: math formula
start = time.time()
total = n * (n - 1) // 2
print(f"Formula: {time.time() - start:.6f}s")
```

The formula is millions of times faster!

This is why **algorithm choice** matters.

Why Sorting Matters

Sorting is EVERYWHERE

"Where did you encounter sorted data today?"

Instagram/TikTok
posts by time/relevance

Spotify
songs by artist/date

Amazon
products by price

Email
messages by date

Game Leaderboards
players by score

Netflix sorts 200+ million users' watch histories!

The Sorting Race

```python
import random, time

def selection_sort(arr):          # naive technique, genuinely O(n^2)
    arr = arr.copy()
    for i in range(len(arr)):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

small = [random.randint(1, 100000) for _ in range(10000)]
large = [random.randint(1, 100000) for _ in range(100000)]

start = time.time()
selection_sort(small)
print(f"Selection sort on 10,000 items: {time.time()-start:.2f}s")

start = time.time()
sorted(large)                     # Timsort -- a merge sort variant!
print(f"Timsort on 100,000 items:       {time.time()-start:.3f}s")
```

Python's `sorted()` uses **Timsort** — the algorithm we build today!

The Sorting Speed Problem

Items	Selection.S O(n^2)	Merge.S O($n \log n$)
1,000	~1 second	**0.01 sec**
100,000	~3 hours	**1.5 sec**
1 million	**12 DAYS**	**~1 sec**

The Sorting Speed Problem

Items	Selection.S O(n^2)	Merge.S O($n \log n$)
1,000	~1 second	**0.01 sec**
100,000	~3 hours	**1.5 sec**
1 million	**12 DAYS**	**~1 sec**

That's why we need smarter algorithms!

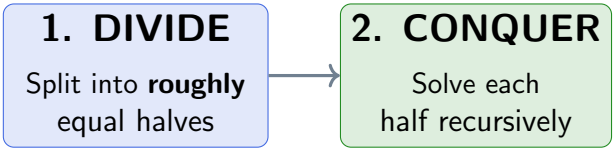
Divide & Conquer Recap

The D&C Recipe

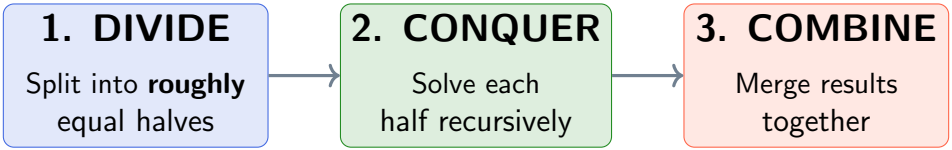
1. DIVIDE

Split into **roughly** equal halves

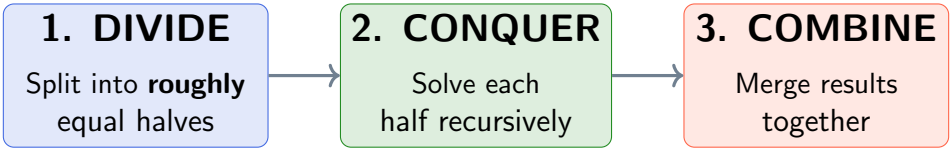
The D&C Recipe



The D&C Recipe



The D&C Recipe



Binary Search: DIVIDE (middle) → CONQUER (one half)
Merge Sort: DIVIDE → CONQUER → **COMBINE**

Binary Search vs. Merge Sort

Step	Binary Search	Merge Sort
DIVIDE	Find middle	Split list in half
CONQUER	Search **one** half	Sort **both** halves
COMBINE	Nothing (just return)	**Merge sorted halves** ← the hard work

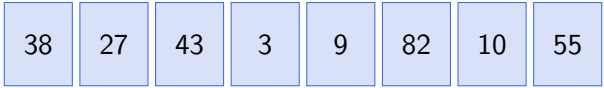
Binary Search vs. Merge Sort

Step	Binary Search	Merge Sort
DIVIDE	Find middle	Split list in half
CONQUER	Search **one** half	Sort **both** halves
COMBINE	Nothing (just return)	**Merge sorted halves** ← the hard work

Binary Search: clever **divide**. Merge Sort: clever **combine**.

The Card Sorting Trick

"How would YOU sort 8 shuffled cards?"



The Card Sorting Trick

"How would YOU sort 8 shuffled cards?"

Compare everything to everything?
Overwhelming!

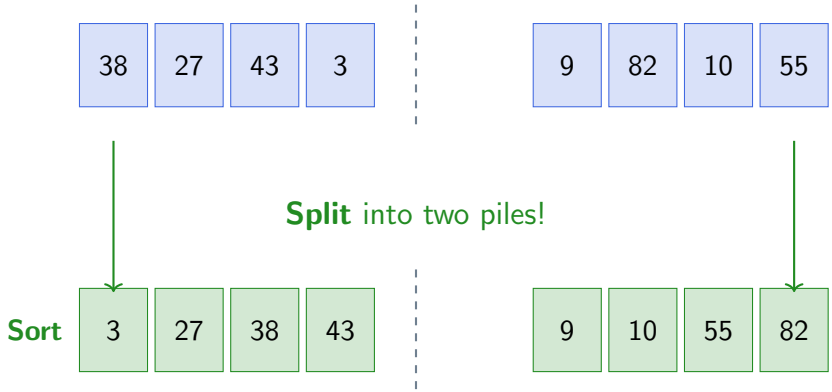
The Card Sorting Trick

"How would YOU sort 8 shuffled cards?"

Split into two piles!

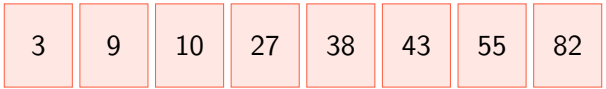
The Card Sorting Trick

"How would YOU sort 8 shuffled cards?"



The Card Sorting Trick

"How would YOU sort 8 shuffled cards?"



Merge into one sorted list!

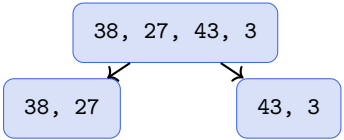
How Merge Sort Works

Merge Sort

38, 27, 43, 3

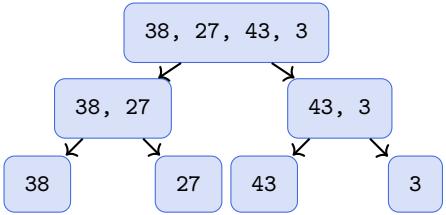
DIVIDE

Merge Sort



DIVIDE

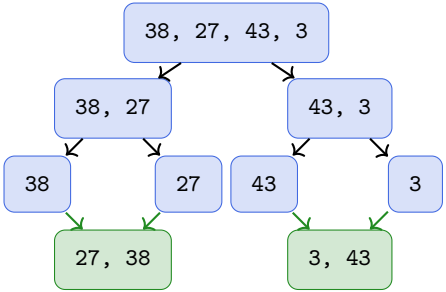
Merge Sort



DIVIDE

Base case!

Merge Sort

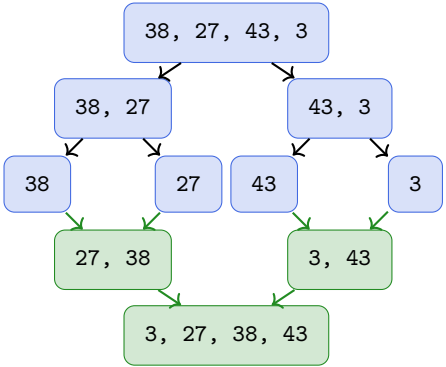


DIVIDE

Base case!

COMBINE

Merge Sort



DIVIDE

Base case!

COMBINE

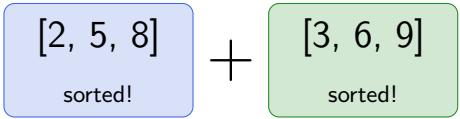
Done!

Step 1: The Dividing Part

```
1  def merge_sort(arr):
2
3      if len(arr) <= 1:                          # BASE CASE: 1 element is already sorted!
4          return arr
5
6      mid = len(arr) // 2
7      left_half = arr[:mid]                      # DIVIDE: Split in half
8      right_half = arr[mid:]
9
10     sorted_left = merge_sort(left_half)        # CONQUER: Recursively sort each half
11     sorted_right = merge_sort(right_half)
12
13     return merge(sorted_left, sorted_right)    # COMBINE: Merge them (next slide!)
```

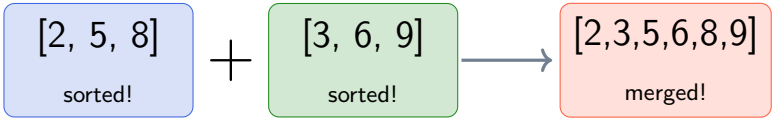
Step 2: The Merge Problem

"Given two SORTED lists, make ONE sorted list"



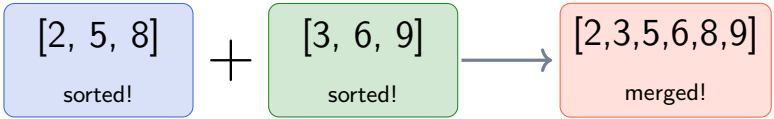
Step 2: The Merge Problem

"Given two SORTED lists, make ONE sorted list"



Step 2: The Merge Problem

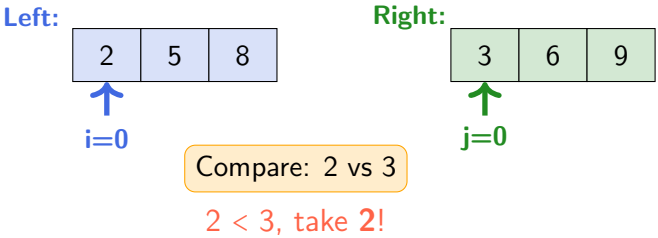
"Given two SORTED lists, make ONE sorted list"



Key Insight: The smallest unmerged item is always at the **front** of one of the two lists!

Merge Step

Use two “fingers” (pointers) to track position in each list:



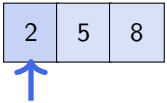
Merge Step

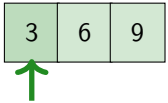
Use two “fingers” (pointers) to track position in each list:

Compare values at both fingers → Take smaller → Move that finger

Merge Step

Step 1:



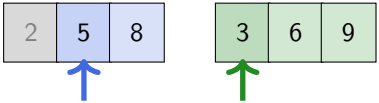


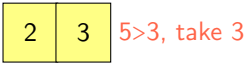
2<3, take 2

Result:

Merge Step

Step 2:



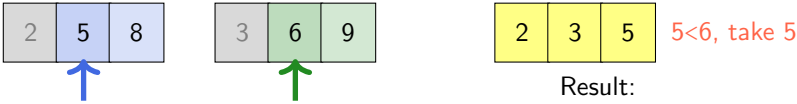


5>3, take 3

Result:

Merge Step

Step 3:



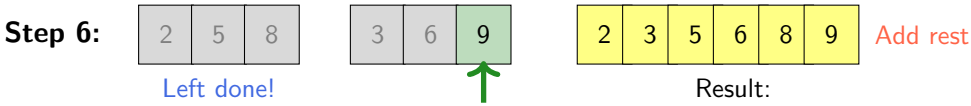
Merge Step

Step 4:

Merge Step

Step 5:

Merge Step



Merge Step

Done! Result: [2, 3, 5, 6, 8, 9]

6 comparisons for 6 elements = **O(n)**

Each element is looked at exactly once!

Merge: The Algorithm

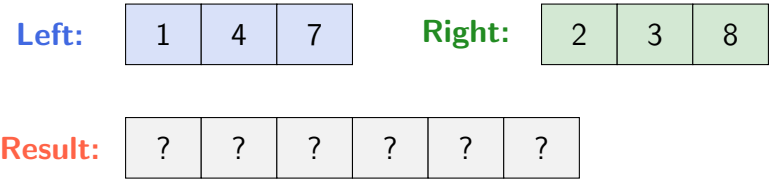
1. Start with pointers at beginning of both lists
2. Compare elements at both pointers
3. Take the **smaller** one, add to result

Merge: The Algorithm

1. Start with pointers at beginning of both lists
2. Compare elements at both pointers
3. Take the **smaller** one, add to result
4. Move that pointer forward
5. Repeat until one list is empty
6. Add all remaining elements from other list

You Try

Merge these two sorted lists by hand:



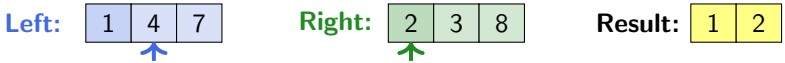
Fill in the table:

Step	Compare	Take	Move
1	1 vs 2	?	?
2	? vs ?	?	?

Solution: Merge [1,4,7] + [2,3,8]

Step	i	j	Compare	Take	Result
1	0	0	1 vs 2	1	[1]

Solution: Merge [1,4,7] + [2,3,8]

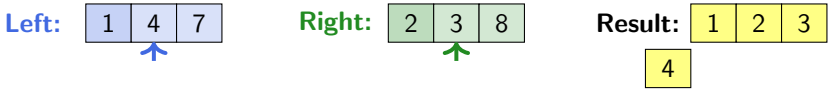


Step	i	j	Compare	Take	Result
1	0	0	1 vs 2	1	[1]
2	1	0	4 vs 2	2	[1,2]

Solution: Merge [1,4,7] + [2,3,8]

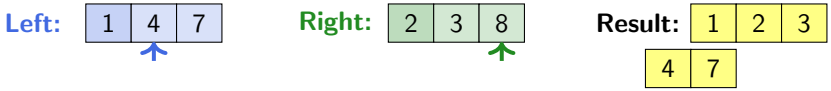
Step	i	j	Compare	Take	Result
1	0	0	1 vs 2	1	[1]
2	1	0	4 vs 2	2	[1,2]
3	1	1	4 vs 3	3	[1,2,3]

Solution: Merge [1,4,7] + [2,3,8]



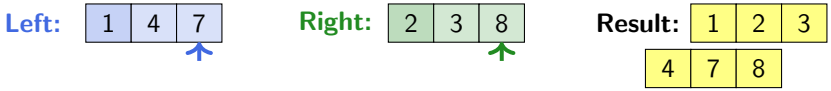
Step	i	j	Compare	Take	Result
1	0	0	1 vs 2	1	[1]
2	1	0	4 vs 2	2	[1,2]
3	1	1	4 vs 3	3	[1,2,3]
4	1	2	4 vs 8	4	[1,2,3,4]

Solution: Merge [1,4,7] + [2,3,8]



Step	i	j	Compare	Take	Result
1	0	0	1 vs 2	1	[1]
2	1	0	4 vs 2	2	[1,2]
3	1	1	4 vs 3	3	[1,2,3]
4	1	2	4 vs 8	4	[1,2,3,4]
5	2	2	7 vs 8	7	[1,2,3,4,7]

Solution: Merge [1,4,7] + [2,3,8]



Step	i	j	Compare	Take	Result
1	0	0	1 vs 2	1	[1]
2	1	0	4 vs 2	2	[1,2]
3	1	1	4 vs 3	3	[1,2,3]
4	1	2	4 vs 8	4	[1,2,3,4]
5	2	2	7 vs 8	7	[1,2,3,4,7]
6	3	2	left done	8	**[1,2,3,4,7,8]**

Step 2: The Merge Code

```
def merge(left, right):
    result = []
    i = 0  # Pointer for left
    j = 0  # Pointer for right

    # Compare fronts, take smaller
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Add remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Complete Merge Sort

```
1  def merge(left, right):
2      ...
3
4  def merge_sort(arr):
5      if len(arr) <= 1:
6          return arr
7      mid = len(arr) // 2
8      left = merge_sort(arr[:mid])
9      right = merge_sort(arr[mid:])
10     return merge(left, right)
```

Test it!

`merge_sort([38,27,43,3])`

→ `[3,27,38,43]`

Quick Check

1. What is the base case?
2. What do we divide?
3. Where is the hard work?
4. What is the Big-O?

Quick Check

1. **What is the base case?**

   A list of 0 or 1 elements — already sorted.

2. **What do we divide?**

3. **Where is the hard work?**

4. **What is the Big-O?**

Quick Check

1. **What is the base case?**

   A list of 0 or 1 elements — already sorted.

2. **What do we divide?**

   The list in half: `mid = len(arr) // 2`

3. **Where is the hard work?**

4. **What is the Big-O?**

Quick Check

1. **What is the base case?**

   A list of 0 or 1 elements — already sorted.

2. **What do we divide?**

   The list in half: `mid = len(arr) // 2`

3. **Where is the hard work?**

   The `merge()` step — combining two sorted halves.

4. **What is the Big-O?**

Quick Check

1. **What is the base case?**

   A list of 0 or 1 elements — already sorted.

2. **What do we divide?**

   The list in half: `mid = len(arr) // 2`

3. **Where is the hard work?**

   The `merge()` step — combining two sorted halves.

4. **What is the Big-O?**

   $O(n \log n)$ — we will see exactly why next.

Tracing Merge Sort

Merge Trace

`merge([27, 38], [3, 43])`:

Step	i	j	Compare	Take	Result
1	0	0	27 vs 3	3	[3]

Merge Trace

`merge([27, 38], [3, 43])`:

Step	i	j	Compare	Take	Result
1	0	0	27 vs 3	3	[3]
2	0	1	27 vs 43	27	[3,27]

Merge Trace

`merge([27, 38], [3, 43])`:

Step	i	j	Compare	Take	Result
1	0	0	27 vs 3	3	[3]
2	0	1	27 vs 43	27	[3,27]
3	1	1	38 vs 43	38	[3,27,38]

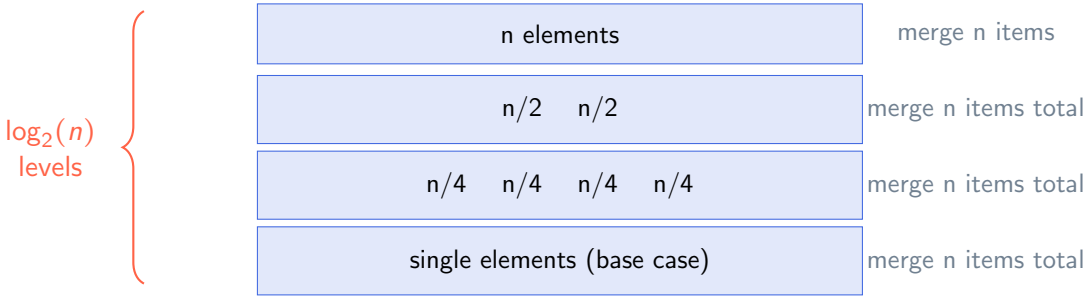
Merge Trace

`merge([27, 38], [3, 43])`:

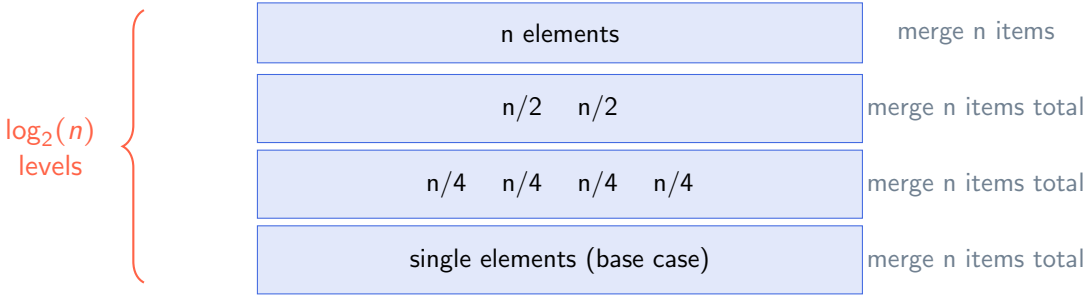
Step	i	j	Compare	Take	Result
1	0	0	27 vs 3	3	[3]
2	0	1	27 vs 43	27	[3,27]
3	1	1	38 vs 43	38	[3,27,38]
4	2	1	left done	43	[3,27,38,43]

Why Is Merge Sort Fast?

The "Levels" Explanation



The "Levels" Explanation



Each level does **n work** $\times$ **log(n) levels** $=$ **O(n log n)**

$O(n^2)$ vs $O(n \log(n))$

n	n^2 (Selection)	$n \log(n)$ (Merge)
100	10,000	**700**
10,000	100,000,000	**133,000**
1,000,000	1,000,000,000,000	**20,000,000**
Speedup:		**50,000x faster!**

The Trade-Off

Pros

Very fast: $On\log(n))$
Stable sorting
Predictable performance

Cons

Uses extra memory
(creates new lists)

The Trade-Off

Pros

Very fast: $O n \log(n))$
Stable sorting
Predictable performance

Cons

Uses extra memory
(creates new lists)

Python's built-in sort() uses a merge sort variant!
(Called `Timsort`)

D&C Beyond Sorting

D&C on Recursive Data

So far: D&C on **flat lists**.

But some data is *naturally* recursive:

Nested folders	JSON / API	HTML / XML
(file system)	responses	documents

If the data is recursive, recursion is the natural tool.

Example 1: Search a Nested Dict

```
config = {
    "app": {
        "name": "MyApp",
        "database": {
            "host": "localhost",
            "port": 5432,
            "credentials": {"user": "admin", "password": "secret123"}
        }
    },
    "debug": True
}

def find_key(data, target_key):
    for key, value in data.items():
        if key == target_key:           # Found at this level!
            return value
        if isinstance(value, dict):     # Recurse deeper
            result = find_key(value, target_key)
            if result is not None:
                return result
    return None

print(find_key(config, "port"))         # 5432
print(find_key(config, "password"))     # "secret123"
```

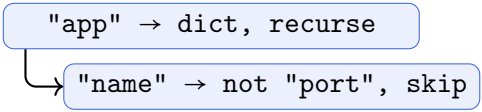
Tracing `find_key`

`find_key(config, "port")`:

`"app"` → `dict, recurse`

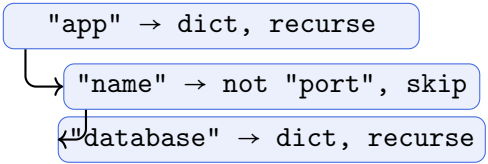
Tracing `find_key`

`find_key(config, "port")`:



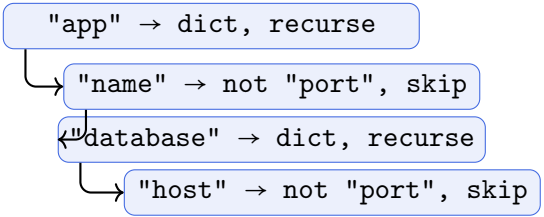
Tracing `find_key`

`find_key(config, "port")`:



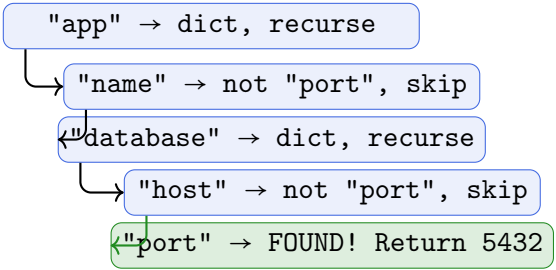
Tracing `find_key`

`find_key(config, "port")`:



Tracing `find_key`

`find_key(config, "port")`:



A loop can't handle this — you don't know how deep it goes.
Recursion handles **arbitrary depth** naturally.

Example 2: File System Size

```
file_system = {
    "name": "root",
    "files": [("readme.txt", 5), ("setup.py", 3)],
    "subfolders": [
        {"name": "src",
         "files": [("main.py", 10), ("utils.py", 7)],
         "subfolders": [
             {"name": "tests",
              "files": [("test_main.py", 8)], "subfolders": []}
         ]},
        {"name": "docs",
         "files": [("guide.md", 15)], "subfolders": []}
    ]
}

def total_size(folder):
    size = sum(f[1] for f in folder["files"])   # files here
    for subfolder in folder["subfolders"]:      # recurse into each
        size += total_size(subfolder)
    return size

print(total_size(file_system))                  # 48
```

Unpacking `total_size`

```
1  def total_size(folder):
2      size = sum(f[1] for f in folder["files"])
3      for subfolder in folder["subfolders"]:
4          size += total_size(subfolder)
5      return size
```

Unpacking `total_size`

```
1  def total_size(folder):
2      size = sum(f[1] for f in folder["files"])
3      for subfolder in folder["subfolders"]:
4          size += total_size(subfolder)
5      return size
```

"Where's the base case?"

It's **implicit**: when a folder has no subfolders, the `for` loop runs **zero** times.
We just return the size of the files — that *is* the base case.

Unpacking `total_size`

```
1  def total_size(folder):
2      size = sum(f[1] for f in folder["files"])
3      for subfolder in folder["subfolders"]:
4          size += total_size(subfolder)
5      return size
```

"Where's the base case?"

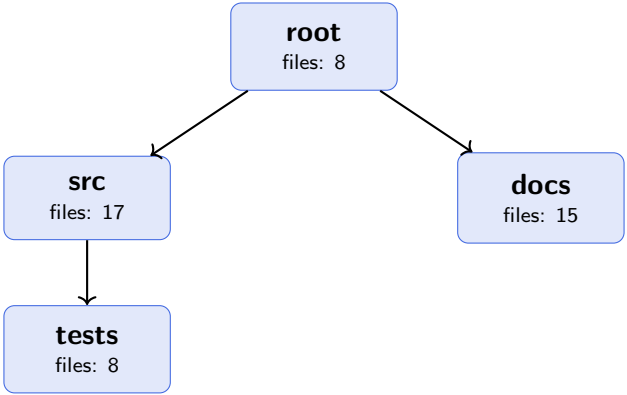
It's **implicit**: when a folder has no subfolders, the `for` loop runs **zero** times.
We just return the size of the files — that *is* the base case.

"There's a for loop — isn't this iterative?"

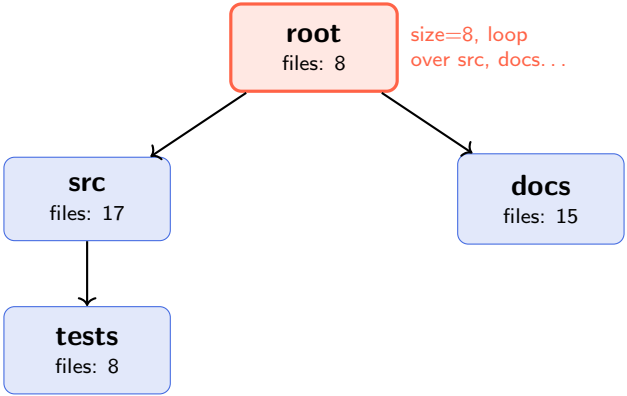
The loop iterates over **siblings** (subfolders at the same level).
Inside the loop, `total_size(subfolder)` **recurses** deeper.

Loop = across siblings, Recursion = down into children.

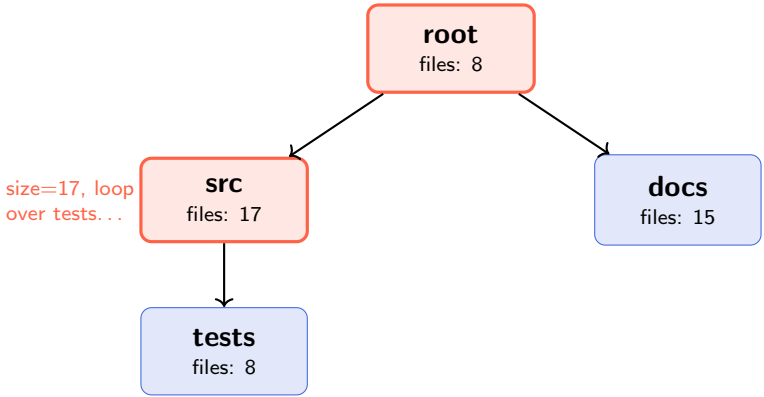
Tracing `total_size`



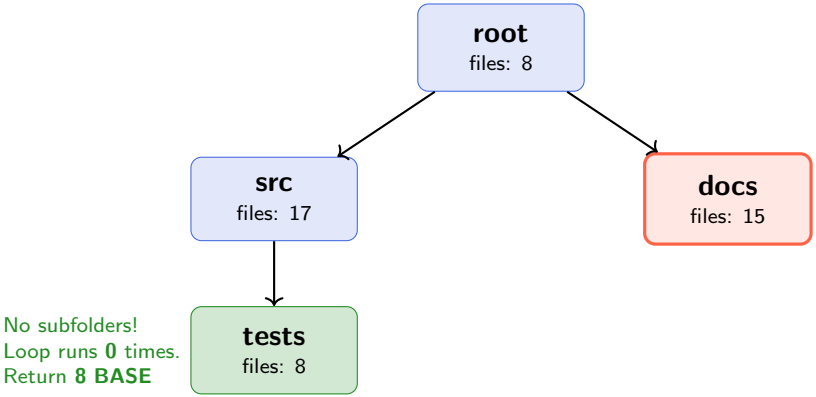
Tracing `total_size`



Tracing `total_size`

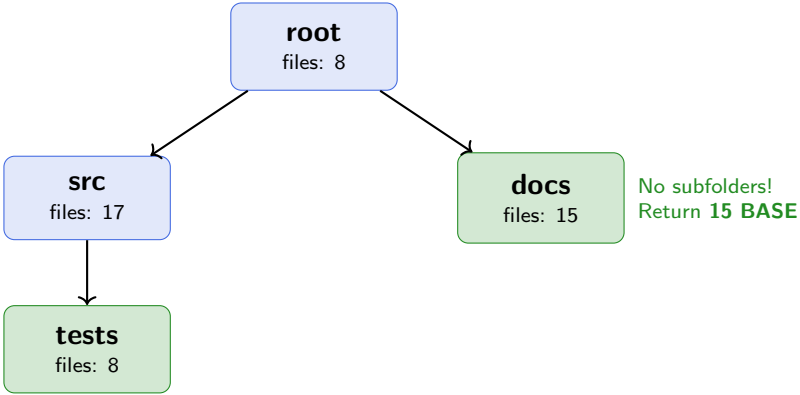


Tracing `total_size`



Base case = leaf folder: `subfolders` is empty ⇒ `for` loop does nothing

Tracing `total_size`



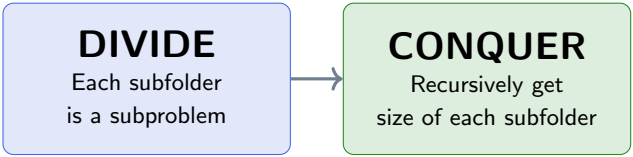
Base case = leaf folder: `subfolders` is empty ⇒ `for` loop does nothing

`total_size` is D&C!

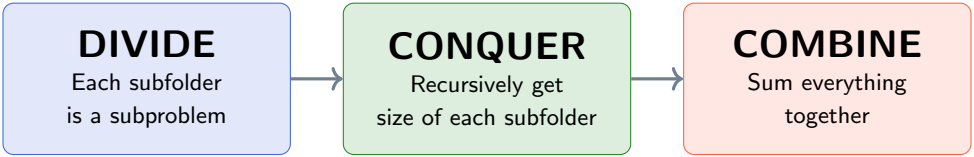
DIVIDE

Each subfolder
is a subproblem

`total_size` is D&C!



`total_size` is D&C!



This is how the `du` (disk usage) command on Linux actually works!

You Try: `count_files`

Fill in the blanks — same pattern as `total_size`:

```
def count_files(folder):
    count = ______________________
    for subfolder in folder["subfolders"]:
        count += _________________

    return count

# count_files(file_system)  -->  ???
```

You Try: `count_files`

Fill in the blanks — same pattern as `total_size`:

```
def count_files(folder):
    count = ______________________
    for subfolder in folder["subfolders"]:
        count += _________________

    return count

# count_files(file_system)  -->  ???
```

Hint: `total_size` summed `f[1]`. What do you count here?

Solution: count_files

```
1  def count_files(folder):
2      count = len(folder["files"])          # files here
3      for subfolder in folder["subfolders"]:
4          count += count_files(subfolder) # recurse
5      return count
6
7  print(count_files(file_system))           # 5
```

Same D&C pattern — just **count** instead of **sum sizes**.

Practice

Merge Sort from Memory

Close your notes. Write merge sort from scratch.

You have **5 minutes**. *Attempt is what matters!*

Merge Sort from Memory

Close your notes. Write merge sort from scratch.

You have **5 minutes**. *Attempt is what matters!*

After 5 minutes, check:

- Who got the **base case** right?
- Who got the **divide step** right?
- Who struggled with the **merge function**?

The merge step is the hardest part.
Focus your study there.

Summary

Today's Key Takeaways

- **Merge Sort** uses Divide & Conquer:
  - DIVIDE: Split list in half
  - CONQUER: Recursively sort each half
  - COMBINE: Merge the sorted halves (the clever part)

Today's Key Takeaways

- **Merge Sort** uses Divide & Conquer:
  - DIVIDE: Split list in half
  - CONQUER: Recursively sort each half
  - COMBINE: Merge the sorted halves (the clever part)
- **$O(n \log n)$** beats $O(n^2)$:
  - n work per level $\times$ $\log n$ levels

Today's Key Takeaways

- **Merge Sort** uses Divide & Conquer:
  - DIVIDE: Split list in half
  - CONQUER: Recursively sort each half
  - COMBINE: Merge the sorted halves (the clever part)
- **$O(n \log n)$** beats $O(n^2)$:
  - n work per level $\times$ $\log n$ levels
- **D&C works on recursive data too:**
  - Nested dicts, file systems, JSON, HTML/XML
  - If the data is recursive, **recursion is the natural tool**

The D&C Template

```
def solve(problem):
    # Base case
    if is_simple(problem):
        return direct_solution
    # Divide
    left, right = split(problem)
    # Conquer
    left_ans = solve(left)
    right_ans = solve(right)
    # Combine
    return merge(left_ans, right_ans)
```

Homework

Part 1: Implement & Trace (Required)

- Implement merge sort from scratch (no notes!), test on `[64,34,25,12,22,11,90]`
- Add `print` statements showing each split and merge

Homework

Part 1: Implement & Trace (Required)

- Implement merge sort from scratch (no notes!), test on `[64,34,25,12,22,11,90]`
- Add `print` statements showing each split and merge

Part 2: Merge Sort Variations (Required)

- Descending merge sort — only *one comparison changes*!
- `is_sorted(arr)` using D&C: check halves, then boundary

Homework

Part 1: Implement & Trace (Required)

- Implement merge sort from scratch (no notes!), test on `[64,34,25,12,22,11,90]`
- Add `print` statements showing each split and merge

Part 2: Merge Sort Variations (Required)

- Descending merge sort — only *one comparison changes*!
- `is_sorted(arr)` using D&C: check halves, then boundary

Part 3: Recursive Data Structures (Required)

- `count_files(folder)` — total file count in nested folder
- `find_all_keys(data, key)` — *list* of all matching values

Homework

Part 1: Implement & Trace (Required)

- Implement merge sort from scratch (no notes!), test on `[64,34,25,12,22,11,90]`
- Add `print` statements showing each split and merge

Part 2: Merge Sort Variations (Required)

- Descending merge sort — only *one comparison changes*!
- `is_sorted(arr)` using D&C: check halves, then boundary

Part 3: Recursive Data Structures (Required)

- `count_files(folder)` — total file count in nested folder
- `find_all_keys(data, key)` — *list* of all matching values

Part 4: Challenge (Optional Bonus)

- Merge K sorted lists using D&C
- Count inversions by modifying merge sort

Next Week: Memoization & Backtracking

Two powerful extensions of recursion:

Memoization

Remember answers you've already computed

Backtracking

Explore possibilities, undo bad choices

Next Week: Memoization & Backtracking

Two powerful extensions of recursion:

Memoization

Remember answers you've already computed

Backtracking

Explore possibilities, undo bad choices

Memoization makes Fibonacci **instant**.

Backtracking solves mazes, Sudoku, N-Queens.

Questions?

Remember: Split → Sort → Merge!