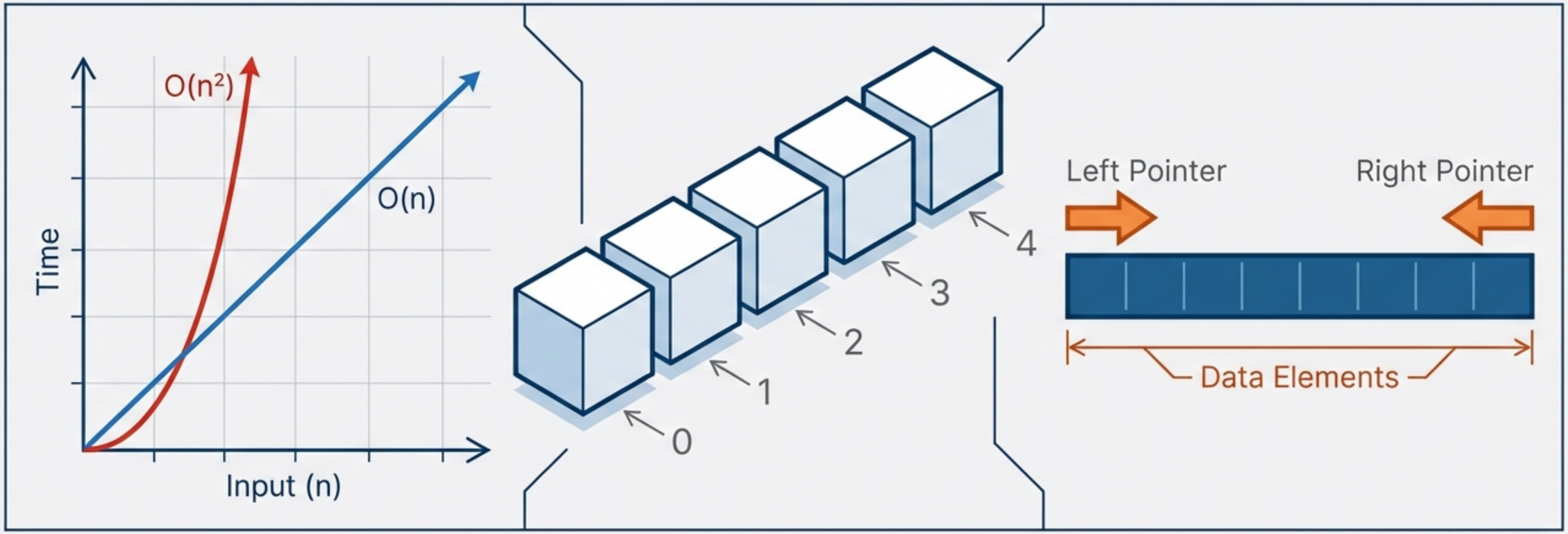
Foundations of Algorithmic Thinking

Mastering Complexity, Python Arrays, and the Two Pointers Pattern



Phase 1: Analysis

NotebookLM

The Ingredient, The Cup, and The Recipe

DATA

Inter Tight

The raw material. Information in its unorganized form.

DATA STRUCTURE

Inter Tight

The container. How we organize and store data for efficient access (e.g., Arrays, Linked Lists).

ALGORITHM

Inter Tight

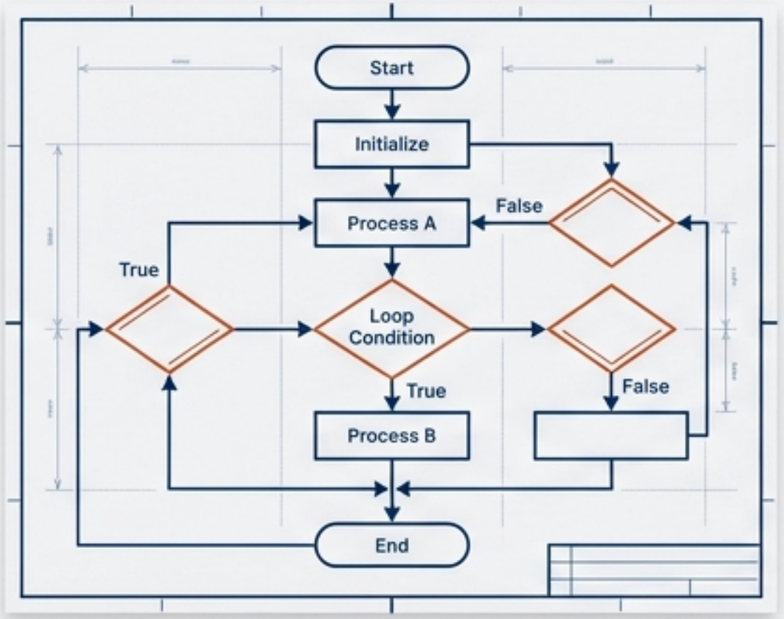
The process. The step-by-step procedure to transform raw materials into a result.

Insight: Software requires both structure (the container) and algorithms (the recipe) to process data effectively.

NotebookLM

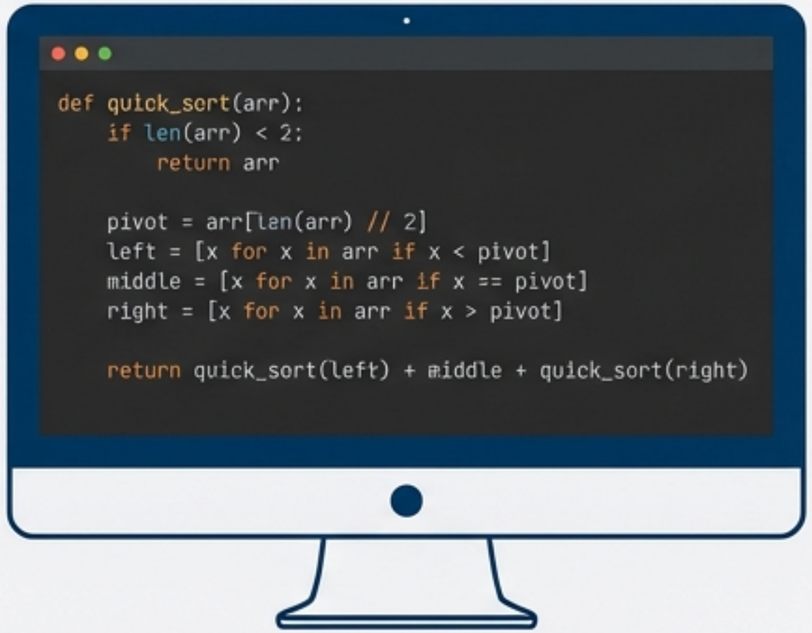
The Design Phase vs. The Implementation Phase

ALGORITHM (Design)



- **Nature:** Mathematical notation or plain English.
- **Dependency:** Independent of hardware, OS, or language.
- **Evaluation:** Priori Analysis (Theoretical function of input size).

PROGRAM (Implementation)



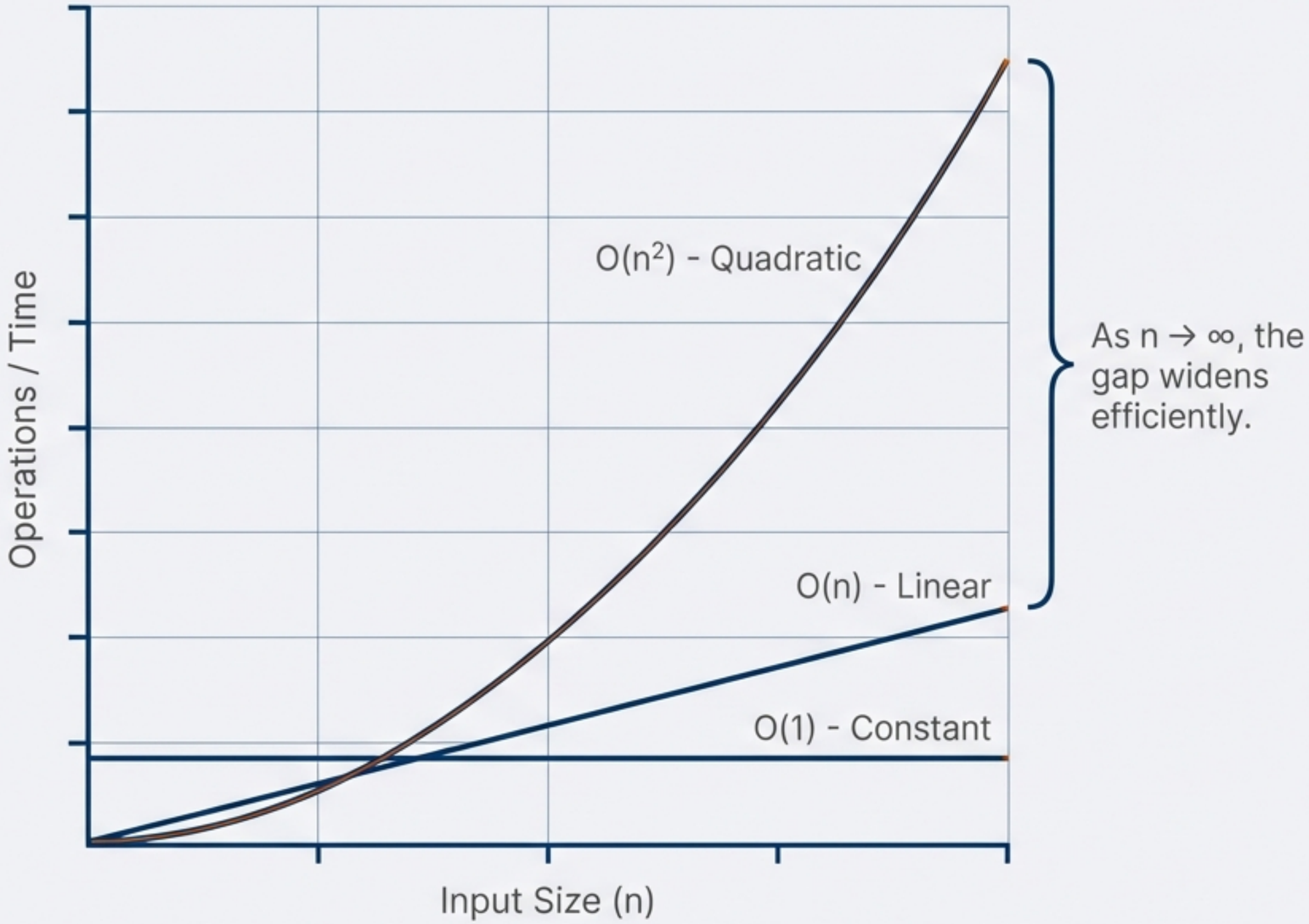
```python
def quick_sort(arr):
    if len(arr) < 2:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)
```

- **Nature:** Specific syntax (Python, C++, Java).
- **Dependency:** Dependent on hardware speed and compiler.
- **Evaluation:** Posteriori Testing (Empirical execution time).

We **analyze algorithms theoretically** because a fast computer can hide the flaws of **a bad algorithm**—until the input gets large.

NotebookLM

Measuring Growth, Not Seconds



The Hierarchy of Efficiency

1. `O(1)`: Constant (Best)
2. `O(log n)`: Logarithmic (Divide & Conquer)
3. `O(n)`: Linear (Loops)
4. `O(n log n)`: Linear-Logarithmic
5. `O(n²)`: Quadratic (Nested Loops)
6. `O(2ⁿ)`: Exponential (Worst)

Concept Box

We focus on the **Dominating Term**. In a function like $2n^2 + 3n + 5$, only n^2 matters when n is large.

NotebookLM

Decoding the Code: Calculating Big O

Constant Time

```
x = 10
y = 20
result = x + y
```

O(1) - No loops, simple arithmetic.

Linear Time

```
for i in range(n):
    print(i)
```

O(n) - Loop runs 'n' times.

Quadratic Time

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

O(n²) - Loop inside a loop ($n \times n$).

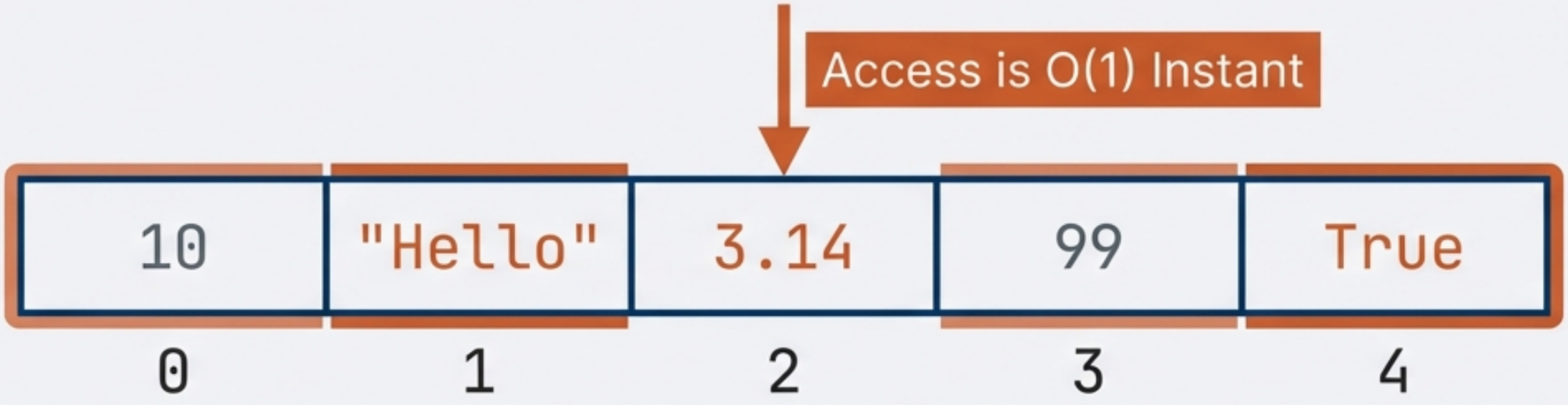
Rule of Thumb

Identify the bottleneck. If you have consecutive loops ($O(n) + O(n)$), it simplifies to $O(n)$. If you have nested loops, they multiply.

NotebookLM

The Container: Python Arrays

Phase 2: Data Structures



Linear & Contiguous

Conceptually stored in a sequence. Elements are accessible via their index.

The Python Twist

Unlike C or Java arrays which are fixed-size and typed, Python Lists are **Dynamic** (auto-growing) and **Heterogeneous** (mixed types).

NotebookLM

Three Flavors of Arrays in Python

The Native List

```
my_list = [1, "Hi", 3.14]
```

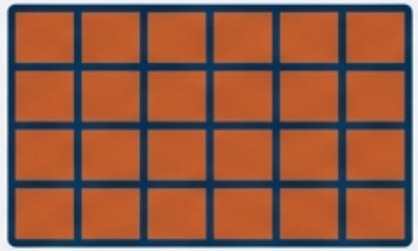
Pros:

- Built-in, extremely flexible, holds mixed data types.

Cons:

- Memory heavy. Slow for massive numerical computations.

The 'array' Module



```
import array
```

Pros:

- Memory efficient. Enforces a single data type (Typed).

Cons:

- Less convenient than Lists. Restricted to homogeneous data.

NumPy

```
import numpy as np
```

Pros:

- The Data Science standard. Multi-dimensional, optimized math operations, creating ranges (`linspace`).

Use Case:

- The powerhouse for scientific computing.

NotebookLM

Python's Superpower: Slicing

```
arr = [10, 20, 30, 40, 50, 60]
```

Subset

`arr[2:5]`

→ [30, 40, 50]

Reversal

`arr[::-1]`

→ [60, 50, 40, 30, 20, 10]

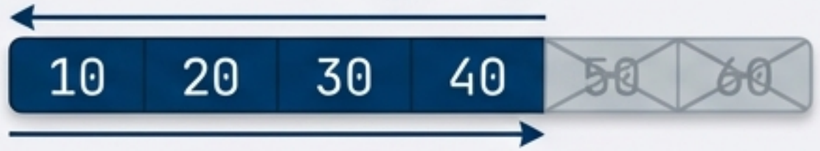
Step / Skip

`arr[::2]`

→ [10, 30, 50]

Negative / Clip

`arr[:-2]`



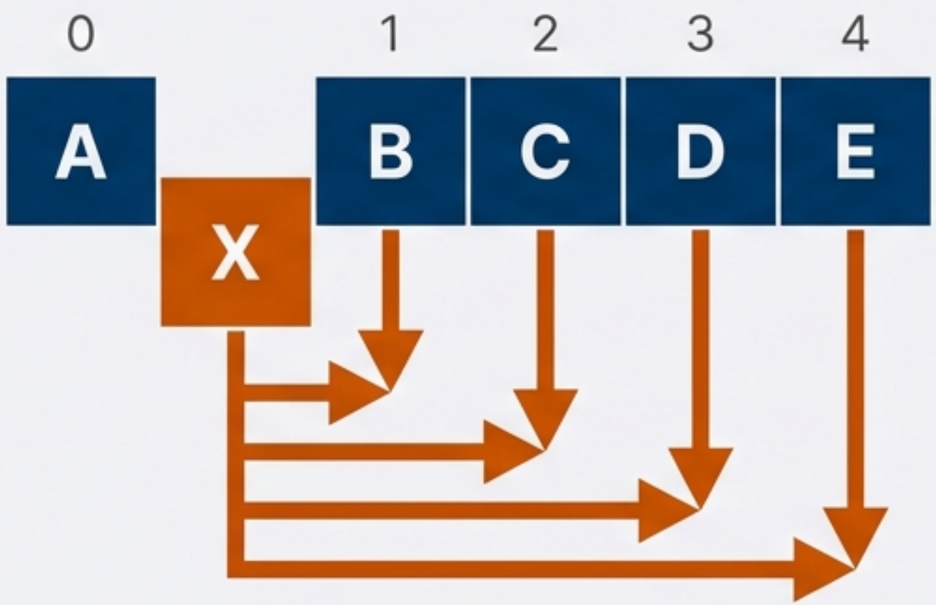
→ [10, 20, 30, 40]

```
array[ start : end : step ]
```

(Start is inclusive, End is exclusive)

NotebookLM

The Cost of Modification



Shifting requires touching every subsequent element.

Operation	Syntax	Complexity	Why?
Append	`append(val)`	**O(1)**	Adds to the end. No shifting needed.
Pop (End)	`pop()	**O(1)**	Removes from end. No shifting.
Insert	`insert(idx, val)	**O(n)**	Elements must shift to create space.
Remove	`remove(val)	**O(n)**	Search + Shift to fill the gap.

Insertion and deletion in the middle of a linear structure costs linear time.

NotebookLM

The Strategy: Two Pointers Pattern

Phase 3: Optimization Patterns

The Problem:
Nested loops ($O(n^2)$) compare every element with every other element. This is often too slow.

The Solution:
Two indices (pointers) traverse the structure simultaneously, processing data in a single pass ($O(n)$).

PCS - Use Two Pointers for:
Partitioning (Separating data)
Comparisons (Checking pairs)
Searching (Finding target values)

NotebookLM

Pattern A: Opposite Ends

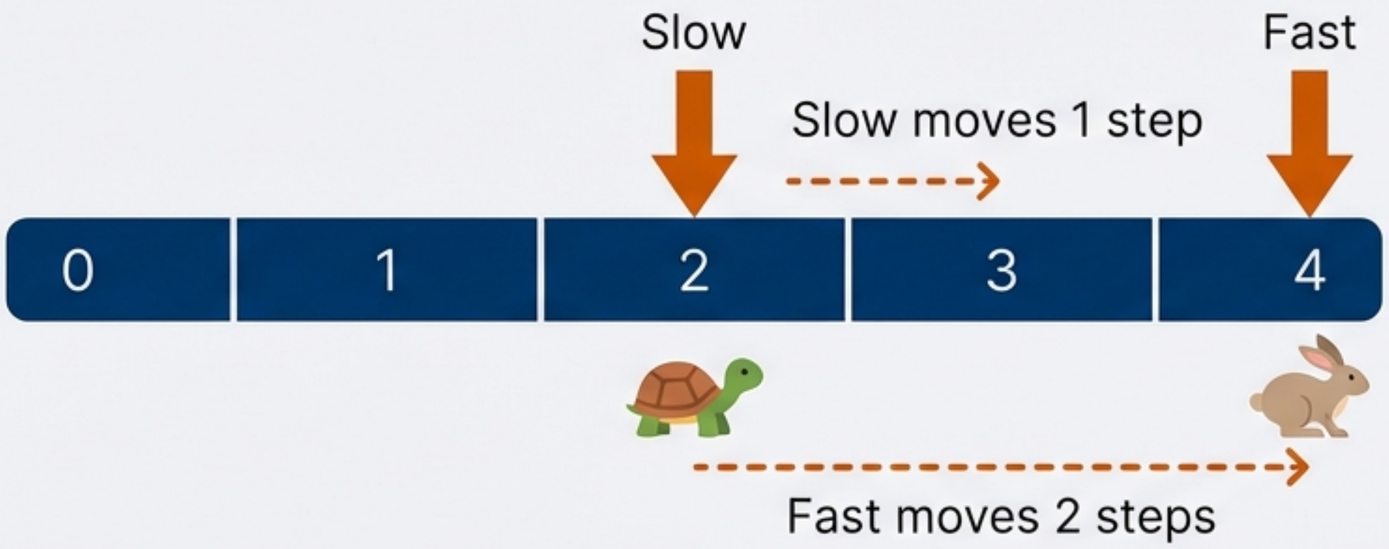
```
left = 0
right = len(array) - 1

while left < right:
    if condition_met:
        return True
    elif value < target:
        left += 1     # Move in from left
    else:
        right -= 1    # Move in from right
```

Use Cases:

- Palindrome Checking
- Two Sum (Sorted Array)
- Reversing an Array in-place

NotebookLM

Pattern B: Fast & Slow Pointers




```
Initialize `Slow = 0`, `Fast = 0`

Iterate:
   `Slow` moves +1,
   `Fast` moves +2.
```

Use Cases:

- Cycle Detection: If Fast catches up to Slow, there is a loop.
- Middle Element: When Fast reaches the end, Slow is exactly at the middle.
- Duplicate Removal: One pointer reads, the other writes unique values.

NotebookLM

Case Study: Palindrome Check

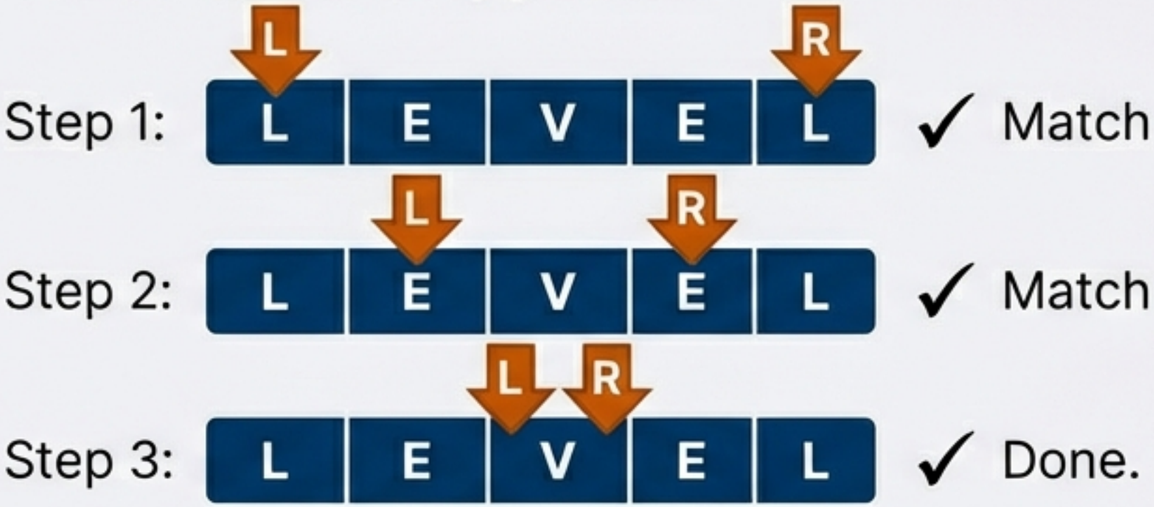
Problem: Is the string "LEVEL" a palindrome?

Naive Approach

Space: O(n)

Requires allocating new memory for the reversed string.

Two Pointers Approach



Space: O(1)

In-place check. No new memory created.

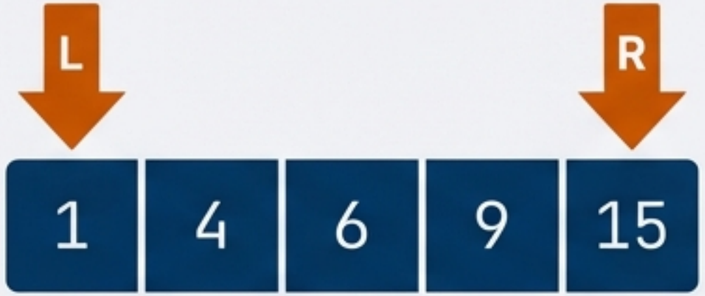
Verdict: Two Pointers saves memory by avoiding data duplication.

NotebookLM

Case Study: Target Sum in Sorted Array

Find two numbers in `[1, 4, 6, 9, 15]` that sum to **24**.

Step 1



Calculation: $1 + 15 = 16$.
Decision: $16 < 24$. Sum is too small. **Move Left Pointer.**

Step 2

Calculation: $\$4 + 15 = 19$.
Decision: $\$19 < 24$. Sum is too small. **Move Left Pointer.**

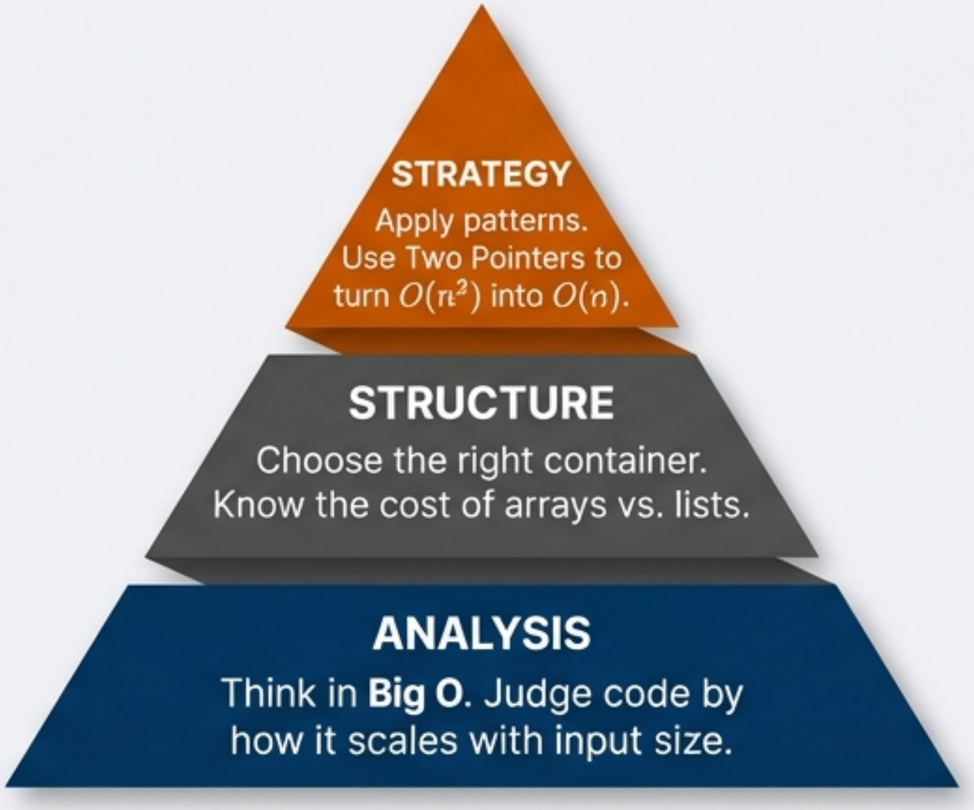
Step 3

Calculation: $9 + 15 = 24$.
Decision: **Target Found.** ✓

Complexity Note:
Brute Force: Try all pairs ($O(n^2)$).
Two Pointers: Make smart decisions based on sorted property ($O(n)$).

NotebookLM

The Algorithmic Mindset



"Efficient code isn't just about getting the right answer. It's about getting the right answer using the least amount of resources.

Next Steps: Practice '**PCS**' problems (Partitioning, Comparisons, Searching) on LeetCode or HackerRank.

NotebookLM