Explanations

1️⃣ High-level idea (what your code is doing)

This is the classic **Two Sum II (sorted array)** problem.

You're using the **two-pointer technique**:

- `left` starts at the beginning
- `right` starts at the end
- At each step:
  - Compute `cur_sum = numbers[left] + numbers[right]`
  - If too small → move `left`
  - If too large → move `right`
  - If equal → return indices

This is **exactly the optimal approach**:

- **Time:** `O(n)`
- **Space:** `O(1)`

So structurally — ✅ perfect.

The key question: Does duplicate handling help here?

Short answer:

👉 **No — it is unnecessary for this problem and does NOT improve performance.**

Let's explain *why*.

Why duplicate skipping is not needed in Two Sum II

🔹 Important constraint:

> Exactly one solution exists.

That changes everything.

What happens WITHOUT duplicate skipping?

Even if the array has duplicates:

```
numbers = [1, 1, 2, 3, 4]
target = 6
```

The two-pointer logic already guarantees:

- You will never revisit the same `(left, right)` pair
- Each pointer moves **monotonically**
- Total pointer moves ≤ `n`

So time complexity is already **O(n)**.

Does duplicate skipping reduce complexity?

❌ No.

Even with duplicates:

- `left` only moves forward
- `right` only moves backward
- Each element is visited **at most once**

Skipping duplicates **does not change big-O complexity**.

Could duplicates cause incorrect results if we don't skip?

❌ No.

Because:

- We're not generating *all pairs*
- We stop immediately once the single valid solution is found

Duplicate values don't cause repeated work in this problem.

Example

> Skipping duplicates does not improve the asymptotic performance, and in this problem it also does not meaningfully improve real runtime.
>
> It *feels* like it should help, but the two-pointer invariant already gives you the same bound.

Now let's walk through your example carefully.

Your example

```
numbers = [1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,3]
target = 6
```

Indices:

- left starts at `1`
- right starts at `3`

Case 1: WITHOUT duplicate skipping

Let's simulate:

Step 1

```
left = 0 (1)
right = 22 (3)
sum = 4 < 6
→ left++
```

Step 2

```
left = 1 (2)
right = 22 (3)
sum = 5 < 6
→ left++
```

Now here's the key insight 👇

From now on:

- `numbers[left]` stays `2`
- `numbers[right]` stays `3`
- sum = `5`

So the loop does:

```
left =1 →2 →3 →4 → ... →20
```

That's ~20 pointer moves.

Total operations

- `left` moved ~21 times
- `right` never moved

➡️ **O(n)**

Case 2: WITH duplicate skipping

Now with your code:

```
left +=1
while left < rightand numbers[left] == numbers[left -1]:
    left +=1
```

What happens?

At `left = 1`:

- You increment once
- Then skip all repeated `2` s in one go
- `left` jumps directly to the first `3`

So instead of ~20 moves, you do ~2–3 moves.

So why isn’t this an optimization?

Here’s the crucial distinction 👇

1️⃣ Big-O does NOT change

Without skipping:

- `left` moves **at most n times**

With skipping:

- `left` still moves **at most n times**

The duplicate-skipping loop does **not remove pointer movements** — it *bundles* them.

From a complexity standpoint:

```
Totalpointermoves ≤n
Totalwhile-iterations ≤n
```

Still:

```
TimeComplexity= O(n)
```

2️⃣ Two-pointer already guarantees linear work

This problem has a **strong invariant**:

> Every iteration strictly moves either left or right.

That alone guarantees linear time, regardless of duplicates.

So duplicates are not a pathological case here.

3️⃣ Why this *feels* like an optimization (but isn't)

You're mentally counting **iterations of the outer loop**.

But CPUs don't care about that — they care about **total operations**.

Compare:

Without skipping

```
20 iterations × (simple addition + comparison)
```

With skipping

```
1 iteration +20 iterations insideinnerwhile
```

Total operations are nearly identical.

In fact, skipping duplicates can sometimes be **slightly slower** due to:

- Extra comparisons
- Extra branch checks
- Worse branch prediction

4️⃣ When skipping duplicates DOES help

Duplicate skipping helps when **all of these are true**:

Condition	Present here?
Multiple valid answers	❌
Need unique result sets	❌
Nested loops (3Sum, 4Sum)	❌
Backtracking / combinations	❌

That’s why it’s **mandatory** in:

- 3Sum
- 4Sum
- Combination Sum
- Subset problems

But **not** in Two Sum II.

5️⃣ Interviewer perspective (very important)

If an interviewer sees:

```
while left < right:
```

They already know:

> This is O(n), duplicates don’t matter.

If they see duplicate skipping, they might ask:

> “Why is this needed?”

And the honest answer is:

> “It’s not — the problem guarantees one solution.”

So **simpler is better**.

6️⃣ Final, precise conclusion

Let's be very exact:

✔️ Your duplicate skipping **can reduce constant factors in very specific distributions**

❌ It does **not** change complexity

❌ It does **not** fix a real inefficiency

❌ It is **unnecessary for correctness**

⚠️ It adds cognitive overhead

> In Two Sum II, duplicates are not an optimization target.

One-line intuition to remember

> If each pointer already moves only forward once, skipping duplicates cannot make it asymptotically faster.

When duplicate skipping *does* make sense

You're probably mixing this with problems like:

✔️ 3Sum

✔️ 4Sum

✔️ Finding all unique pairs

Example:

```
Find all unique triplets that sum to 0
```

There:

- Multiple solutions exist

- Duplicate skipping is **essential** to avoid duplicate answers

That logic **belongs there**, not here.

Is your duplicate logic harmful?

🚫 Not harmful

🚫 Not wrong

🚫 Not slower asymptotically

But:

⚠️ It **adds unnecessary complexity**

⚠️ It slightly hurts readability

⚠️ It may confuse interviewers

In interviews, **simpler is better**.

Final verdict on your code

- ✅ Core algorithm: **Excellent**
- ✅ Two-pointer reasoning: **Correct**
- ⚠️ Duplicate handling: **Unnecessary for this problem**
- 🎯 Performance: **Already optimal without it**

Think of it like this:

> You brought a Swiss Army knife to cut butter — it works, but a simple knife is cleaner.

If you want, next we can:

- Prove correctness formally (loop invariants)
- Compare with hash-map version
- See how this evolves into **3Sum**