Reduce without initial accumulator

```
const longest = arr.reduce((acc, curr) =>
  curr.length > acc.length ? curr : acc
);
console.log(longest);
```

if we are not adding acc in dependency than what happens like in this we don't aded any initial acc

Great question, Vipul!
Let's break this down **in a very beginner-friendly way**.

✅ What happens if we do NOT give an initial accumulator (acc)?

When you write:

```
js

const longest = arr.reduce((acc, curr) =>
  curr.length > acc.length ? curr : acc
);
```

Here, you **did NOT pass an initial value** to `reduce`.

👉 So what does JavaScript do?

JavaScript automatically sets:

- `acc` = **first element** of the array
- `curr` = starts from the **second element**

Example:

```
js

const arr = ["apple", "banana", "kiwi"];
```

Internally, reduce does:

Iteration	acc	curr	output
start	"apple"	—	(no compare)
1	"apple"	"banana"	"banana"
2	"banana"	"kiwi"	"banana"

Final result → `"banana"`

❗ What happens if the array is empty?

If the array is empty:

```
js

[].reduce(...)
```

and you **did not** pass an initial accumulator...

👉 JavaScript will throw an error:

```
sql

TypeError: Reduce of empty array with no initial value
```

💡 Why does this work for longest string?

Because for longest string:

- First element is a string
- Comparison `curr.length > acc.length` works fine

So Javascript can use the **first element** as `acc`.

⭐ Should you give an initial value?

It is safer to ALWAYS give an initial value:

```
js

const longest = arr.reduce(
  (acc, curr) => (curr.length > acc.length ? curr : acc),
  "" // initial empty string
);
```

Benefits:

✔️ Works even for empty arrays
✔️ Makes the logic more readable
✔️ Avoids runtime errors

🧠 Summary (super simple)

- **No initial value → first element becomes acc**

- **curr starts from second element**
- **Empty array → error**
- Works fine only when array has at least one element

If you want, I can also explain this using a dry run or small animation-style diagrams.

> and we are not returning anything for here than why it is working

Great question, Vipul!
You're noticing something important:

> **"We are not writing `return` inside the reduce callback... then how is it working?"**

Let's clear this in a beginner-friendly way. 👇

✅ Reason: Arrow functions can return values implicitly

When you write a function like this:

```
js

(curr.length > acc.length ? curr : acc)
```

This is called an **implicit return**.

👉 In arrow functions:

- If you use **no curly braces `{}`**
- Then whatever expression you write is **automatically returned**

Example:

```
js

const add = (a, b) => a + b;
```

Even though there is no **`return`**, it returns **`a + b`**.

💡 Your reduce code uses implicit return

```
js

arr.reduce((acc, curr) =>
  curr.length > acc.length ? curr : acc
);
```

Here:

- No `{ }`
- So the expression after `=>` is returned automatically

This:

```
js

curr.length > acc.length ? curr : acc
```

is **returned** implicitly.

Same as writing:

```
js

arr.reduce((acc, curr) => {
  return curr.length > acc.length ? curr : acc;
});
```

⭐ If you write curly braces `{}`, then return is REQUIRED

This WILL NOT work:

```
js

arr.reduce((acc, curr) => {
  curr.length > acc.length ? curr : acc;  // ❌ NO RETURN → undefined
});
```

Because arrow functions with `{}` do **NOT** return automatically.

🧠 Conclusion

✔️ It works because the arrow function uses **implicit return**.
✔️ Without `{}`, JavaScript automatically returns the expression.
✔️ With `{}`, you must write `return`.

Want a simple rule?

> **Arrow function with no `{}` → auto return**

Arrow function with `{}` → must write return manually

If you want, I can show visual diagrams for both or rewrite it in simple JS without reduce.