1. useMemo , useCallback and memo

useMemo -

Purpose: To **memoize (remember)** a computed value so it **doesn't re-calculate unnecessarily**.

When to use:

- When a calculation is **heavy or slow**
- And its inputs **don't change often**

Purpose (उद्देश्य):

`useMemo` किसी भी **computed value** को याद **(memoize)** करके रखता है, ताकि वह बार-बार दोबारा **calculate** न हो।

कब उपयोग करें?

- जब कोई calculation बहुत **heavy** या **slow** हो
- और उसकी dependency (input values) ज़्यादा बार नहीं बदलती हों

Example

```
const expensiveValue = useMemo(() => {
  return slowCalculation(num);
}, [num]);
```

✔ `slowCalculation` will **only run when `num` changes**,
❌ NOT on every re-render.

useCallback -

Purpose: To memoize a function so that React reuses the same function instance between renders.

Why is it needed?

- In React, functions are recreated on every render.
- If you pass a function as a prop to a child component,
  the child may re-render even if it's not needed.

Purpose (उद्देश्य):

`useCallback` किसी **function** को **memoize** करता है, ताकि React हर **render** पर नया **function** न बनाए।

क्यों ज़रूरी है?

React में हर बार render होने पर functions नया बन जाते हैं।
अगर आप ऐसे functions को **props** के रूप में **child component** को भेजते हैं, तो child component अनावश्यक रूप से **re-render** हो सकता है, भले उसका data न बदले।

Example
```
const handleClick = useCallback(() => {
  setCount(count + 1);
}, [count]);
```

✔ `handleClick` will remain the same reference
❌ until `count` changes.

Result:

✔ `handleClick` का reference वही रहता है
❌ जब तक `count` न बदले

👉 इसलिए child को लगता है कि function prop वही है, इसलिए वो re-render नहीं करता

memo -

Purpose: To prevent re-rendering of a functional component unless props change.

Think of it as **PureComponent for function components.**

Purpose (उद्देश्य):

`memo` किसी **functional component** को अनावश्यक **re-render** होने से रोकता है, जब तक उसके **props change** न हों।

इसे आप **functional components** के लिए **PureComponent** जैसा समझ सकते हैं।

Example
```
const Child = React.memo(function Child({ value }) {
```

```
  console.log("Rendered");
  return <div>{value}</div>;
});
```

✔ Child re-renders **only when `value` changes**,
❌ Not when parent re-renders unnecessarily.

Result:

✔ Child component सिर्फ तभी re-render होगा जब `value` बदले
❌ Parent का render होने पर भी child re-render नहीं होगा

How They Work Together

These three often work as a team:

`memo()` → stops child component re-render **when props don't change**

`useCallback()` → ensures function props **don't change**

`useMemo()` → ensures calculated values **don't change**

So memoization only works correctly if both values and functions stay stable.

Quick Summary Table

Hook / API	Memoizes	Prevents?
`useMemo`	**Value**	**Expensive recalculation**
`useCallback`	**Function**	**Re-creating functions**
`memo`	**Component Rendering**	**Unnecessary child re-render**

When NOT to use Them

❌ Don't overuse memoization
❌ If calculation is cheap → don't use `useMemo`
❌ If no child depends on stable functions → don't use `useCallback`

❌ If component is small and fast → skip `memo`

They help performance only when there are real re-render costs.

Example Without Optimization (BAD PERFORMANCE)

Parent re-renders → Child re-renders even if props don't change
Because function props get recreated every render.

```
function Parent() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState("");

  const updateText = (newText) => {
    setText(newText);
  };

  return (
    <>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>

      <Child text={text} updateText={updateText} />
    </>
  );
}

function Child({ text, updateText }) {
  console.log("Child rendered");
  return (
    <div>
      <input value={text} onChange={e => updateText(e.target.value)}
/>
    </div>
  );
}
```

Problem

- Clicking **Increment** in Parent causes **Child to re-render**

- Even though `text` has not changed
- Because `updateText` function was recreated on every Parent render

Optimized Version Using memo + useCallback

Goal

- **Prevent Child re-render when only Parent count changes**

Step 1 — Memoize the Child

```
const Child = React.memo(function Child({ text, updateText })
{
  console.log("Child rendered");
  return (
    <div>
      <input value={text} onChange={e =>
           updateText(e.target.value)} />
    </div>
  );
});
```

✔ Now Child will **only re-render if props change**

Step 2 — Memoize the Function using useCallback

```
function Parent() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState("");

  const updateText = useCallback((newText) => {
    setText(newText);
  }, []); // function remains stable!

  return (
    <>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <Child text={text} updateText={updateText} />
    </>
  );
}
```

✔ Now what happens?

- **Parent re-renders when count changes**
- **BUT Child:**
    - **receives same `text`**

    - **and same `updateText` function**

- **So Child does NOT re-render**

How useMemo works in the same scenario

Let's say you have a heavy calculation:

```
const expensiveValue = useMemo(() => {

    return expensiveCalculation(text);

}, [text]);
```

✔ `expensiveCalculation` runs **only when text changes**,
❌ NOT when count changes.

Then pass it to Child:

```
<Child

  text={text}

  updateText={updateText}

  calculatedValue={expensiveValue}

/>
```

No extra re-renders because:

- `memo` protects child
- `useCallback` keeps function stable
- `useMemo` keeps heavy value stable

What You Should Observe in Console

Try running this example.

- Increment count → Parent re-renders
- **Child does NOT log "Child rendered"**
- Child re-renders only when:
    - You type in input (text changes)
    - Or if dependency for `updateText` changes

So memoization **saved at least 50–90% useless renders** depending on UI.

2. useMemo & useEffect mai difference kya hota h

useMemo — Memoization (VALUE RETURN करता है**)**

क्या करता है?

- कोई **expensive calculation** बार–बार दोबारा न चले
- React **render** होने से पहले ही value मेमोराइज़ कर लेता है

```
const result = useMemo(() => {
  return heavyCalculation(a, b);
}, [a, b]);
```

Key Points:

✔ **render** के दौरान चलता है (side-effect नहीं करता)
✔ एक **VALUE return** करता है
✔ Value तभी दुबारा calculate होती है जब **dependency** बदले
❌ DOM update या API call नहीं करता
❌ asynchronous नहीं है

useMemo = Computation Optimization

useEffect — Side Effects (किसी **VALUE** को नहीं लौटाता**)**

क्या करता है**?**

- **React render** के बाद **side effects execute** करता है जैसे**:**
    - **API calls**
    - **DOM update**
    - **localStorage update**
    - **timers, subscriptions**

```
useEffect(() => {
  fetchData();
}, [id]);
```

Key Points:

✔ **render** के बाद चलता है (post-render)
✔ **side effects** करता है
✔ cleanup भी कर सकता है
✔ asynchronous code allow करता है

useEffect = Side Effect Handling

5 सेकंड में सबसे बड़ा **difference**

Feature	**useMemo**	**useEffect**
कब चलता है**?**	**render** के दौरान	**render** के बाद
क्या **return** करता है**?**	एक **value**	कुछ नहीं
किसलिए**?**	**heavy calculation optimize** करने के लिए	**side effects** चलाने के लिए
allows async?	नहीं	हाँ
DOM/API call?	नहीं	हाँ
cleanup?	नहीं	हाँ

Real-World Example to Understand

❌ Problem:

अगर आप **expensive calculation** लिखें**:**

const value = heavyCalculation(a, b);

तो यह हर **render** पर दोबारा चलेगा → **slow UI**

Solution with useMemo:

✔ calculation सिर्फ a या b बदलने पर चलेगी

✔ बाक़ी renders में cached value वापस मिलेगी

When to use useEffect?

जब आपको API call करनी हो:

```
useEffect(() => {
  fetch(`api/user/${id}`);
}, [id]);
```

❌ इस काम के लिए useMemo कभी नहीं use करना
क्योंकि **useMemo** सिर्फ **calculation optimization** है, कोई effect या async action नहीं चलाता

Magic line for Interviews

useMemo = compute and cache a value during render.
useEffect = run side effects after render.

Case: हम एक भारी **calculation** और एक **API call** करना चाहते हैं

❌ अगर दोनों को बिना **hooks** चलाएँ:

```
function App({ id }) {
  // heavy calculation (हर render में दोबारा चलेगी)
  const value = heavyCalculation(id);

  // API call (हर render में चली जाएगी) ❌ BAD
  fetch(`/api/user/${id}`);

  return <div>{value}</div>;
}
```

Problem:

- हर बार render → value दोबारा calculate हो जाती है
- हर बार render → API दोबारा call हो जाती है
- UI slow + server load बढ़ा

अब यही चीज़ हम **useMemo** और **useEffect** से **optimize** करेंगे

useMemo वाला पार्ट **(Heavy Calculation)**

```
function App({ id }) {
  const value = useMemo(() => {
    console.log("Heavy calculation running...");
    return heavyCalculation(id);
  }, [id]);

  return <div>{value}</div>;
}
```

Behaviour:

- जब `id` बदले → calculation दोबारा चलेगी
- जब parent re-render हो जाए पर `id` same रहे → calculation नहीं चलेगी
- यह **render** के दौरान चलता है
- यह **value return** करता है

✔ Perfect optimization for expensive math, loops, filtering, sorting, etc.

useEffect वाला पार्ट **(API Call + Side Effects)**

```
function App({ id }) {
  const value = useMemo(() => {
    console.log("Heavy calculation running...");
    return heavyCalculation(id);
  }, [id]);

  useEffect(() => {
    console.log("API call running...");
    fetch(`/api/user/${id}`);
  }, [id]);

  return <div>{value}</div>;
}
```

Behaviour:

- जब `id` बदले → API दोबारा call होगी
- जब `id` same रहे और केवल parent re-render हो जाए → API नहीं चलेगी
- यह **render** के बाद चलता है
- कोई value return नहीं करता

✔ Perfect for:

- API request
- DOM manipulation
- subscriptions
- timers
- cleanup logic

Console Output Sequence (MOST IMPORTANT)

मान लो `id` बदला

Console होगा:

Heavy calculation running... // useMemo (render के दौरान)

API call running... // useEffect (render के बाद)

अगर parent re-render हुआ लेकिन `id` नहीं बदला:

(no logs at all)

क्योंकि**:**

- cached memo value वापिस मिल गया
- effect dependency नहीं बदली, इसलिए API नहीं चली

Thus → **zero unnecessary work!**

Where Many Devs Get Confused

कभी-कभी लोग पूछते हैं:

> useMemo और useEffect दोनों dependencies देखते हैं, तो दोनों same क्यों नहीं हैं?

Answer is:

- **useMemo value cache** करता है
- **useEffect side effects** चलाता है

और सबसे बड़ा फर्क:

> **useMemo = synchronous computation**
> **useEffect = asynchronous side-effect after paint**

> **ONE-LINE INTERVIEW ANSWER**
>
> *useMemo heavy calculations* को *memoize* करता है *during render* और *value return* करता है।
> *useEffect render* के बाद *side effects execute* करता है, कोई *value return* नहीं करता।

3. can we use useEffect instead of useMemo if yes then why we need this and if no then why not?

   ❌ **Short Answer: NO — useEffect cannot replace useMemo**

   क्योंकि दोनों का काम**, timing,** और **return type** अलग है।

Why useEffect ≠ useMemo? (Deep but Simple)

useMemo value return करता है

const result = useMemo(() => heavyCalc(a, b), [a, b]);

result को आप JSX या state में तुरंत use कर सकते हैं

- यह **render** के दौरान चलता है
- कोई side-effect नहीं करता
- सिर्फ **calculate + memoize** करता है

useMemo = calculate and return value

useEffect कोई value return नहीं करता

```
useEffect(() => {
  heavyCalc(a, b); // चल तो जाएगा
}, [a, b]);
```

- यह **render** के बाद चलता है
- value return नहीं करता
- state change करना पड़ेगा result use करने के लिए
- और state change render trigger करेगा
- फिर useEffect दोबारा चलेगा
- then re-render again…

👉 infinite re-render risk ❌

If you try to replace useMemo with useEffect

You would be forced to do this:

```
const [value, setValue] = useState(null);

useEffect(() => {
  const result = heavyCalc(a, b);
  setValue(result);
}, [a, b]);
```

क्यों यह गलत है?

- **render → effect → setState → render again**
- unnecessary extra render
- UI slow
- ही heavy value को calculate करके भी **immediate** नहीं मिलती, क्योंकि effect बाद में चलता है

useEffect = late + extra re-render
useMemo = immediate + no extra render

BIG DIFFERENCE — Timing

Hook	कब चलता है?
useMemo	**render** के दौरान **(immediate)**
useEffect	**render** के बाद **(asynchronous)**

So:

❌ **useEffect** से **UI** को **value** तुरंत नहीं मिल सकती
✔ **useMemo** से **UI** को **value** उसी **render** में मिल जाएगी

Practical Example Why useEffect is Wrong Here

Imagine:

const filteredUsers = heavyFilter(users);

return <UserList data={filteredUsers} />;

अगर आप इसे **useEffect** से करेंगे**:**

- पहले **render** होगा **without filtered data**
- **effect** चलेगा
- फिर **state** अपडेट
- फिर दुबारा **render**
- **UI flickering**

✔ **useMemo** में**:**

- पहली **render** में ही **filtered value** तैयार

So question again: Can we use useEffect instead of useMemo?

Technically YES

आप कर सकते हैं… लेकिन**:**

- **UI** दो बार **render** होगा
- **performance** खराब होगी
- **unnecessary state updates** होंगे
- **computation** और **side-effects mix** हो जाएंगे
- **logic messy** हो जाएगा

इसलिए बिल्कुल नहीं करना चाहिए

WHY WE NEED useMemo AT ALL

Because this gives us:

1. **heavy calculation without extra re-render**
2. **instant return value**
3. **no state management overhead**
4. **clean synchronous logic**

इसे useEffect से replace करना गलत **design pattern** है

MOST IMPORTANT CONCLUSION

> **useMemo = value calculation**
>
> **useEffect = side effect**
>
> **useEffect** कभी भी **useMemo** का **substitute** नहीं है क्योंकि यह **return** नहीं करता, और **state update must use** करेगा, जिससे **extra render** होगा।

Example

हम एक बहुत **heavy calculation** मान लेते हैं (जैसे 1 से 50 लाख तक loop):

```
function heavyCalculation(num) {
  console.log("heavyCalculation running...");
  let total = 0;
  for (let i = 0; i < 5_000_000; i++) {
    total += num;
  }
  return total;
}
```

✅ Version 1: useMemo वाला सही तरीका

```
import React, { useState, useMemo, useEffect } from "react";

function AppUseMemo() {
  const [count, setCount] = useState(0);

  console.log("AppUseMemo render हो रहा है...");

  const computed = useMemo(() => {
    return heavyCalculation(count);
  }, [count]);
```

```
  return (
    <div>
      <h2>useMemo Version</h2>
      <p>Count: {count}</p>
      <p>Computed: {computed}</p>
      <button onClick={() => setCount((c) => c + 1)}>Increment</button>
    </div>
  );
}
```

इस **version** में क्या होगा?

- हर बार जब तुम **Increment** दबाओगे:
    - `AppUseMemo` re-render होगा
    - `heavyCalculation` सिर्फ तब चलेगा जब `count` बदलेगा
- कोई **extra render** नहीं लगेगा
- Calculation उसी **render** के दौरान हो जाती है
- Console में:
    - `AppUseMemo render हो रहा है...`
    - `heavyCalculation running...`
    - बस एक बार per change

Version 2: useEffect से वही काम करने की गलती

```
import React, { useState, useEffect } from "react";

function AppUseEffect() {
  const [count, setCount] = useState(0);
  const [computed, setComputed] = useState(0);

  console.log("AppUseEffect render हो रहा है...");

  useEffect(() => {
    const result = heavyCalculation(count);
    setComputed(result);
  }, [count]);
```

```
  return (
    <div>
      <h2>useEffect Version</h2>
      <p>Count: {count}</p>
      <p>Computed: {computed}</p>
      <button onClick={() => setCount((c) => c + 1)}>Increment</button>
    </div>
  );
}
```

यहाँ क्या होगा?

1. तुमने **Increment** दबाया
2. `count` बदलता है → component **render** होता है
    - Console: `AppUseEffect render हो रहा है...`
3. Render के बाद `useEffect` चलेगा
    - Console: `heavyCalculation running...`
4. `setComputed(result)` → फिर से एक नया **render** trigger होगा
    - Console: `AppUseEffect render हो रहा है...` (दुबारा)

👉 मतलब हर **click** पर **2 renders**
👉 heavy calculation भी extra चक्कर लगा सकती है कुछ scenarios में
👉 UI भी पहले पुराना data दिखाएगी, फिर update होगा (flicker जैसा behavior possible)

🔍 अगर **tum React DevTools Profiler use** करो

- `AppUseMemo`:
    - हर increment पर **1 render**
- `AppUseEffect`:
    - हर increment पर **2 renders** (एक count change से, एक computed state change से)

Same काम करने के लिए useEffect वाली approach में:

- ज्यादा renders
- extra state
- ज़्यादा complexity
- slow UI

इसलिए हम कहते हैं:

> **Computation / derived values → `useMemo`**
> **Side effects (API, DOM, subscription, timers) → `useEffect`**

🎯 Final याद रखने लायक बात

❌ **useEffect** को **useMemo** की जगह **use** कर सकत हो **technically**,
लेकिन वो:

- ज़रूरत से ज़्यादा render कराएगा
- UI को value late देगा
- performance worse करेगा

✅ इसलिए React ने अलग hook दिया है:

- `useMemo`: *"value ko memoize karo, extra render mat* करवाओ*"*
- `useEffect`: *"render ke baad side-effect chalao"*

Interview Answer:

> **No, we should NOT use `useEffect` instead of `useMemo`.**
>
> `useMemo` memoizes a computed value **during render**, so the UI gets the result immediately **without extra renders**.
>
> If we use `useEffect`, we must store the computed result in state, which causes **one render for the calculation and another render for state update**, making the component slower and more complex.

So:

useMemo = compute + return value efficiently
useEffect = run side effects after render (API, DOM, subscriptions)

Therefore, both are fundamentally different and **not interchangeable**.