Hook that can improve performance of the app

**useMemo**

* Cache the results b/t re-renders🡪react cannot re-calculate the expensive operation again and again🡪we don’t want to perform heavy operation again and again
* Calculation part is heavy operation like prime number calculation
* With memo
  + When we try to change other state variable the heavy calculation also recalculate again and it sometimes freeze the browser when calculation is slow
  + useMemo 🡪 cache the calculation and return the result

**🔍 Given:**

**useEffect(() => findtext(text), [text]);**

**useMemo(() => findtext(text), [text]);**

Assume text is a state variable updated via setText.

**🤔 Do they run the same number of times?**

**Not exactly.** Even though both depend on [text], **they are used for different purposes**, and they behave differently:

**✅ useEffect(() => findtext(text), [text])**

* **Runs AFTER render**.
* Triggers **every time text changes**.
* Used for **side effects** (API calls, logging, DOM access, etc).
* You **can’t return a value** (except a cleanup function).

So if text changes 5 times, it calls findtext 5 times **after** rendering.

**✅ useMemo(() => findtext(text), [text])**

* **Runs DURING render**, before DOM is painted.
* Used to **memoize a computed value**.
* findtext(text) is **evaluated only if text changes**.
* It **returns a value** you use in the render.

So if you do:

const result = useMemo(() => findtext(text), [text]);

findtext(text) will run only when text changes. It **returns a memoized value**, not a side-effect.

**🔁 Frequency-wise:**

If you're comparing purely the number of **times findtext is *called***:

* Both are **called the same number of times**, i.e., when text changes.
* BUT they run in **different phases**:
  + useMemo → during render (pure)
  + useEffect → after render (side-effect)

**🔥 Summary Table:**

| **Feature** | **useEffect** | **useMemo** |
| --- | --- | --- |
| Purpose | Side-effects | Memoizing expensive computation |
| Timing | After render | During render |
| Return Value | (Optional) Cleanup function | Memoized value |
| findtext use | For API/logging/side-effects | For returning processed data |

can we use setstate in usememo?

Great question!

### ⚠️ Short answer:

**You should NOT use setState inside useMemo.**

### ❓ Why not?

Because useMemo is meant for **pure computations**, and **should not cause side effects** like setting state. It runs **during render**, and calling setState during render can lead to:

* **React warnings**
* **Unexpected behavior**
* **Infinite re-renders** (boom! 💣)

Is useMemo pure function?

**Yes, useMemo is meant to be used with a *pure function*.**

**🔬 What’s a *pure function*?**

A function is **pure** if:

* It **always returns the same output** for the same input.
* It **does not cause side effects** (like modifying external variables, calling setState, logging, making API calls, etc).

**What happens if it's not pure?**

If your function inside useMemo:

* Calls setState
* Logs things
* Fetches data
* Modifies outside values

…you're misusing it. React may behave unexpectedly, or you’ll see warnings like:

"Cannot update a component while rendering a different component"

| **✅ DO use useMemo for** | **❌ DON’T use useMemo for** |
| --- | --- |
| Expensive calculations | State updates (setState) |
| Filtering/sorting data | API calls or logging |
| Derived values from props/state | Modifying external data |

why useEffect not ideal for heavy calculation?

Awesome follow-up! Here's a crystal-clear explanation of **why useEffect is not ideal for heavy calculations**:

**🚀 useEffect runs after the component is rendered**

That’s the key point.

**🧠 Think of this timeline:**

1. React starts render ➡️

2. UI is painted to screen (user sees it) ➡️

3. useEffect runs (findtext is called here)

**❌ So what’s the problem?**

If findtext(text) is **CPU-heavy** (e.g. filtering 100,000 records, calculating primes, etc.), and it runs in useEffect, then:

* The UI **renders first** — possibly without correct data or placeholder.
* The **expensive task starts afterward**.
* When it finishes, it might **call setState**, causing **another re-render**.

This can lead to:

* **Laggy UI**
* **Flickering/flash of incomplete data**
* **Wasted renders**

**🧪 Example: Filtering a huge dataset**

useEffect(() => {

const result = heavyFilter(text); // takes 500ms

setFiltered(result);

}, [text]);

* User types “a”
* UI renders blank or stale list
* 500ms later → list updates
* If user types again → repeat lag

**✅ Why useMemo is better here**

const filtered = useMemo(() => heavyFilter(text), [text]);

* Computation happens **during render**
* Result is **ready immediately**
* UI **renders once with correct result**
* No extra render or flash

**🔥 TL;DR**

|  | **useEffect** | **useMemo** |
| --- | --- | --- |
| When it runs | After render | During render |
| UI behavior | Might render incomplete data | UI shows computed value instantly |
| Re-renders? | Causes one extra render | No extra render |
| Use for | Side effects (API, logs, DOM) | Pure computations (filters, maps) |

**⚠️ Rule of Thumb:**

**If your logic returns a value and slows down the UI — use useMemo.**  
**If it does a side effect — use useEffect.**

Want to see this with a live interactive example that clearly shows the lag vs smooth UI?

**React.memo** 🡪 it is HOC(higher order component) that helps to avoid re-render of component if its props remains unchanged 🡪 Wrap a component in memo to get a memoized version of that component. This memoized version of your component will usually not be re-rendered when its parent component is re-rendered as long as its props have not changed.

Think of it like this:

🛣️ **React.memo = “skip if nothing changed… unless I have a reason to re-check.”**

**Why is it not a 100% guarantee?**

Because:

1. **React might flush effects or internal updates** that still cause rendering.
2. **Dev tools**, **strict mode**, or certain hooks like useContext can **force re-renders**.
3. **Parent component changes** or **layout updates** may also lead to a re-render under some conditions.
4. **React.memo only does shallow comparison of props — if a prop is an object/array/function that changes on every render, memoization fails unless you use useMemo/useCallback.**

**Paramters to React.Memo**

**1.component that need to memo**

**2.optional arugemnt to be passed for comparision which Is done automatically by react itself to compare each props send to child**

**Const Todo = React.Memo(function Todo(props){**

**Return <div>re-render</div>**

**})**

Or

**Const Todo = React.memo( ({props})=>{ 🡪 pass component itself as props to it**

**Return <div>re-render</div>**

**})**

**Behind screen memo will check prevProps !=props then re-render**

#### Returns

memo returns a new React component. It behaves the same as the component provided to memo except that React will not always re-render it when its parent is being re-rendered unless its props have changed.

Usages:

* Skips unwanted re-render of component
* Updating memorized component using state

It is better to use Memo with useCallback when we need to prevent calling child component 🡪as when we pass function from parent to children and it parent for each re-render new function will be created which will make component re-render

import "./styles.css";

import UserMemo from "./UserMemo";

import { useCallback, useState } from "react";

export default function App() {

  const [email, setEmail] = useState("");

  const [name, setName] = useState("");

  // this calls the re-render of child only when name changees

  const handleClick = useCallback(() => {

    console.log("click");

  }, [name]);

  // const handleClick = () => {

  //   console.log("test");

  // };

  return (

    <div className="App">

      <input

        placeholder="email"

        name="email"

        value={email}

        onChange={(e) => {

          setEmail(e.target.value);

        }}

      />

      <input

        placeholder="name"

        name="name"

        value={name}

        onChange={(e) => {

          setName(e.target.value);

        }}

      />

      {/\* the userMemo will be re-rendered only if the name props changes - it normally render during first render then re-render happen only after change of name \*/}

      <UserMemo name={name} />

      {/\* for functions in memo as it compares shallow copy each time when parent render few function is created so it re-render child too \*/}

      {/\* To avoid above we use useCallback with react.memo \*/}

      <UserMemo name={name} handleClick={handleClick} />

    </div>

  );

}

**useCallback:**

useCallback is a React Hook that lets you cache a function definition between re-renders.

Parameters:

Fn:it takes function that need to be cached🡪callback function

Dependency -> dependency array that need to make function re-render

**Return :**

During initial render it returns new function and on sub sequent render the useCallback returns a new function when dependency changes

Without dependency array as empty the component will re-render and useCallback don’t have effect

**useRef:**

why we do we use it?

* Case where we need to keep data that no need to re-render
* **Want some value but no need to re-render 🡪 can we use constant or local variables?**
* Problem is .. we cannot change it and if we need to change the value we need to use useState which will cause re-render
* **when we change the value of variable it will change the value but it wont re-render**
* **why state variable if we have let and var**
  + The react will not be able to know when there is change in value and will not update the ui with latest value so we use useState to hold previous value and update the ui
* **Why useRef?**
  + When we use state and normal variable in component the variable value will re-set when the state variable changes and re-render the component function and re-intilize the local variable again but we should hold previous value how? 🡪 with useRef 🡪 we need to re-render the component but need to hold the variable
  + **When we use useRef 🡪it will keep the variable value b/t each re-render but we cannot display in ui**
  + **It is like local variable that wont re-render on each component re-render and holds the value**