useMemo Hook

- useMemo is a React hook introduced to optimize the performance of functional components.
- It memoizes the result of a computation and recalculates it only when its dependencies change.
- This can be particularly useful for expensive computations or derived values that do not need to be re-evaluated on every render.
- Think of useMemo as a way to "remember" the result of a function until its inputs change.

Core Concept

React re-renders components whenever their state or props change. During re-renders, all functions within the component are executed, which includes any calculations or derived values.
In cases where calculations are computationally heavy or where re-renders are frequent, this can degrade performance. The useMemo hook solves this problem by caching the result of a computation and recomputing it only when its dependencies change.

When to use useMemo

- When you have costly computations (e.g., sorting, filtering, or performing heavy mathematical operations).
- When you want to avoid recalculations for values derived from props or state that remain unchanged during most renders.
- When you're working with large datasets where recalculating derived values would be wasteful.

Syntax

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

where,

- computeExpensiveValue(a, b): The function to compute the value.
- [a, b]: Dependency array—useMemo will recompute the value only if one of these dependencies changes.

Example

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

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

  // Expensive computation only recalculates when count changes
  const expValue = useMemo(() => {
    console.log("Calculating...");
    return count * 5;
  }, [count]);

  return (
    <div>
      <p>Expensive Value: {expValue}</p>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
    </div>
  );
}
export default App;
```

Why use useMemo?

- **Preventing Expensive Recomputations**: When you have computations that are resource-intensive (e.g., filtering a large list, mathematical calculations, etc.), useMemo ensures the computation is skipped unless necessary.
  Example: A component displays a sorted list. Without useMemo, the sorting operation would run on every render.
- **Avoiding Re-Renders in Derived Data:** Derived data, which depends on other states or props, can cause unnecessary renders. useMemo helps in caching derived state until dependencies change.
- **Improving Performance:** In large applications, frequent renders can degrade user experience. Optimizing calculations with useMemo can reduce this overhead.

Key Characteristics of useMemo

- **Memoized Value:** The value is cached, and React will return the cached value unless the dependencies change.
- **Dependency Array:** Similar to useEffect, the dependency array determines when the computation should re-run.
- **Lazy Evaluation:** The computation function is executed lazily—only when the component renders and the dependencies change.
- **Pure Function:** The function passed to useMemo should not have side effects. It must always return the same output for the same input.

Example

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

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

  // Expensive computation only recalculates when count changes
  const expValue = useMemo(() => {
    console.log("Calculating...");
    return count * 10;
  }, [count]);

  return (
    <div>
      <p>Expensive Value: {expValue}</p>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <input    type="text"    value={text}    onChange={(e) =>
setText(e.target.value)}    placeholder="Type something"
      />    </div>
  );
}
export default App;
```

Behaviour : The expensive computation (count * 10) runs only when count changes, not when text changes.

Example: Managing a Filtered List with Expensive Computation

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

function App() {
  const [count, setCount] = useState(0);
  const [searchTerm, setSearchTerm] = useState("");

  // Large data set of products
  const products = useMemo(() => Array.from({ length: 10000 }, (_, i) => `Product
${i + 1}`), []);

  // Expensive computation to filter items
  const filteredProducts = useMemo(() => {
                    console.log("Filtering products...");
                              return products.filter((product) =>
                              product.toLowerCase().includes(searchTerm.toLowerCase()));
                    }, [searchTerm, products]);
  return (
    <div>
      <h1>Optimized List Filter</h1>
      <p>Counter: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>

      <input        type="text"        placeholder="Search product"
value={searchTerm}        onChange={(e) => setSearchTerm(e.target.value)}      />

      <ul>
       {filteredProducts.slice(0, 20).map((product) => (
                                                  <li key={product}>{product}</li>
       ))}
      </ul>
    </div>
  );
}
export default App;
```

Behavior of useMemo at statement:

// Large data set of products
  const products = useMemo(() => Array.from({ length: 10000 }, (_, i) => Product ${i + 1}), []);

How useMemo works in this context: The function () => Array.from({ length: 10000 }, (_, i) => Product ${i + 1}) creates an array of 10,000 products (e.g., ["Product 1", "Product 2", ..., "Product 10000"]).
The empty dependency array [] tells React that this computation does not depend on any external values or state, so the function will only run once, during the component's initial render.

What happens on subsequent renders: On subsequent renders of the component, React will skip executing the function inside useMemo because the dependencies ([]) have not changed.
Instead, React will reuse the memoized value of the products array from the initial render.

Why use useMemo here: Generating an array of 10,000 products is a computationally expensive operation. If this computation were to run on every render, it could significantly degrade performance.
By using useMemo, React avoids recalculating the products array unnecessarily, improving efficiency.

Key Points: The products array is created only once, no matter how many times the component re-renders.
If the dependency array were to include values that change (e.g., [someDependency]), the computation would re-run whenever someDependency changes.

Analogy

- Think of useMemo as a cache for the computed value (products array). As long as the cache's dependencies remain the same, React will fetch the value from the cache instead of recalculating it.
- This behavior is particularly useful for optimizing components that perform heavy computations or handle large datasets.

Difference Between useCallback and useMemo

Both useCallback and useMemo are React hooks introduced to optimize performance. While they are conceptually similar (both deal with memoization), they serve different purposes and are used in different scenarios.

Aspect	useCallback	useMemo
Purpose	Memoizes a function.	Memoizes a value or computation.
Primary Use Case	Prevents unnecessary re-creation of a function.	Prevents unnecessary re-execution of a computation.
Return Value	Returns a memoized version of the callback function.	Returns the memoized result of a computation or derived data.
Usage Scenarios	Used when passing a function as a prop to child components, especially with React.memo.	Used to avoid recalculating expensive computations.
Dependencies	Recreates the function only when dependencies change.	Recomputes the value only when dependencies change.
Focus	Focuses on optimizing function references.	Focuses on optimizing computation results.
Performance Impact	Prevents child components from unnecessary re-renders due to function prop changes.	Improves performance by skipping unnecessary calculations.
Relationship with React.memo	Works best with React.memo to prevent child re-renders.	Independent of React.memo, but both can be used together for optimization.
Example Return	Returns a callback function: () => {}.	Returns a value: 10, [1, 2, 3], etc.

When to Use useCallback vs useMemo

Scenario	Hook Used
Passing a function as a prop to a child component	useCallback
Avoiding re-renders of child components due to unstable function references	useCallback
Preventing expensive calculations from running unnecessarily	useMemo
Deriving or transforming state that depends on other states/props	useMemo

Example: Using useCallback and useMemo Together

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

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

  // Expensive calculation memoized
  const calculate = useMemo(() => {
    console.log("Running expensive calculation...");
    return count * 10;
  }, [count]);

  // Stable callback function
  const handleIncrement = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  return (
    <div>
      <p>Expensive Calculation: {calculate}</p>
      <button onClick={handleIncrement}>Increment</button>
      <input type="text" value={text} onChange={(e) =>
setText(e.target.value)} placeholder="Type something"
     />
      <Child onClick={handleIncrement} />
    </div>
  );
}

function Child({ onClick }) {
  console.log("Child Component rendered!");
  return <button onClick={onClick}>Call Increment</button>;
}

export default App;
```

Points to understand

- **useMemo for Expensive Calculation:** In the parent component, the expensive computation (count * 10) runs only when count changes. Typing in the input field does not trigger the computation.

```
const calculate = useMemo(() => {
  console.log("Running expensive calculation...");
  return count * 10;
}, [count]);
```

- **useCallback for Stable Function:** The incrementCount function is wrapped with useCallback, ensuring the same function reference is passed to the Child component unless count changes. This prevents unnecessary re-renders of the Child.

```
const handleIncrement = useCallback(() => {
  setCount((prevCount) => prevCount + 1);
}, []);
```

- **Child Component:** The Child component receives the memoized handleIncrement function as a prop. Without useCallback, every re-render of the parent would recreate the function, causing the Child to re-render unnecessarily.

Example: Filtering a List with useMemo and useCallback

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

function App() {
  const [filterText, setFilterText] = useState("");
  const [count, setCount] = useState(0);

 const items = [
    "Chair",   "Table",    "Cooler",    "Fan",    "Tube Light",    "Television",
    "Laptop", "Mobile", "Projector"
  ];

  // Memoized filtered list
  const filteredItems = useMemo(() => {
    console.log("Filtering items...");
    return items.filter((item) =>
      item.toLowerCase().includes(filterText.toLowerCase())
    );
  }, [filterText]);

  // Memoized change handler for filter input
  const handleFilterChange = useCallback((e) => {
    setFilterText(e.target.value);
  }, []);

  // Memoized click handler to increment count
  const incrementCount = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  return (
    <div>
      <h1>Filter Example</h1>
      <input      type="text"      placeholder="Filter items..."       value={filterText}
        onChange={handleFilterChange} // Stable callback
      />
      <button onClick={incrementCount}>Increment Count</button>
      <p>Count: {count}</p>
      <ul>
        {filteredItems.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Combined use of useMemo and React.memo: Imagine a parent component rendering a child list component. The list contains filtered data based on user input, and we want to prevent the child component from re-rendering unnecessarily when unrelated parent state changes.

Example

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

// Memoized Child Component
const Child = React.memo(({ items }) => {
  console.log("Child component re-rendered");
  return (
    <ul>
      {items.map((item, index) => (
                              <li key={index}>{item}</li>
      ))}
    </ul>
  );
});
```

```
function App() {
  const [filterText, setFilterText] = useState("");
  const [count, setCount] = useState(0);

  const items = [
    "Chair",   "Table",   "Cooler",   "Fan",   "Tube Light",   "Television",
    "Laptop", "Mobile", "Projector"
  ];

  // Memoized filtered list
  const filteredItems = useMemo(() => {
    console.log("Filtering items...");
    return items.filter((item) =>
      item.toLowerCase().includes(filterText.toLowerCase())
    );
  }, [filterText]);

  return (
    <div>

      <input
        type="text"
        placeholder="Filter items..."
        value={filterText}
        onChange={(e) => setFilterText(e.target.value)}
      />
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <p>Count: {count}</p>
      <Child items={filteredItems} />
    </div>
  );
}
export default App;
```

Behaviour: The child component is re-rendering repeatedly in your code because the array reference for filteredItems is changing even though the contents of the array might remain the same.

To ensure the child component renders only once, we need to ensure that the props passed to it remain constant after the initial render. This can be achieved by:

Using useMemo to ensure the filtered list does not create a new reference unnecessarily. Wrapping the items array in useMemo or moving it outside the component so it is constant.

Example

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

// Memoized Child Component
const Child = React.memo(({ items }) => {
  console.log("Child component rendered");
  return (
    <ul>
     {items.map((item, index) => (
      <li key={index}>{item}</li>
     ))}
    </ul>
  );
});

function App() {
  const [filterText, setFilterText] = useState("");
  const [count, setCount] = useState(0);

  // Static items array wrapped in useMemo to ensure it stays constant
  const items = useMemo(() => [
    "Chair", "Table", "Cooler", "Fan", "Tube Light", "Television",
    "Laptop", "Mobile", "Projector"
  ], []); // Empty dependency array ensures it doesn't change

  // Memoized filtered list
  const filteredItems = useMemo(() => {
    console.log("Filtering items...");
    return items.filter((item) =>
      item.toLowerCase().includes(filterText.toLowerCase())
    );
  }, [filterText, items]);

  return (
    <div>
      <input
        type="text"
        placeholder="Filter items..."
        value={filterText}
        onChange={(e) => setFilterText(e.target.value)}
      />
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <p>Count: {count}</p>
      <Child items={filteredItems} />
    </div>
  );
}

export default App;
```

Why the Child Renders Only Once?

- The items array reference remains constant due to useMemo.
- The filteredItems reference changes only when filterText changes.
- React.memo ensures the Child component renders only when its props change, which happens only when filteredItems updates.

By ensuring the props to the Child component are stable, unnecessary re-renders are avoided, and the component renders only once (unless genuinely required).

Homework

Use useMemo with:

1. useState: Memoize values derived from the state.
2. useEffect: Minimize dependencies by memoizing computations.
3. useContext: Optimize context values using useMemo.
4. useReducer: Memoize derived values from complex reducer states.
5. useCallback: Understanding the interplay of useMemo and useCallback.