React Memoization: Concept, Syntax, and Working Examples

1. What is Memoization (General Definition)

Memoization is a computing technique that stores the result of an expensive operation and reuses that stored result when the same input occurs again, instead of recalculating it.

Key ideas: - Save previous results - Avoid repeating the same work - Trade memory for speed - Works best when inputs repeat frequently

---

2. What is Memoization in React?

In React, memoization is a performance optimization strategy used to avoid unnecessary re-renders, recalculations, and function recreations during component updates.

React provides three primary memoization tools: - **React.memo** – memoizes components - **useMemo** – memoizes values - **useCallback** – memoizes functions

These tools do not change application behavior; they only improve performance.

---

3. React.memo — Component Memoization

Definition (Interview Ready)

- Wraps a functional component
- Performs a shallow comparison of props
- Skips re-render when props are the same
- Useful for pure, prop-driven components

Syntax

```
const MemoizedComponent = React.memo(function Component(props) {
  return <div>{props.value}</div>;
});
```

Working Example

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

const Counter = React.memo(({ count }) => {
  console.log("Counter rendered");
  return <h2>Count: {count}</h2>;
});

export default function App() {
  const [count, setCount] = useState(0);
  const [toggle, setToggle] = useState(false);

  return (
    <div>
      <Counter count={count} />
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setToggle(!toggle)}>Toggle UI</button>
      {toggle && <p>UI toggled</p>}
    </div>
  );
}
```

What happens: - Clicking **Increment** → Counter re-renders (prop changed) - Clicking **Toggle UI** → Counter does NOT re-render (props same)

---

4. useMemo — Value Memoization

Definition (Interview Ready)

- Takes a function and dependency array
- Stores the computed value in memory
- Recalculates only when dependencies change
- Used for expensive calculations

Syntax

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

Example 1 — Expensive Calculation

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

const sumArray = (arr) => arr.reduce((a, b) => a + b, 0);

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

  const numbers = useMemo(() =>
    Array.from({ length: 1000000 }, (_, i) => i),
    []
  );

  const sum = useMemo(() => sumArray(numbers), [numbers]);
  return (
    <div>
      <h2>Sum: {sum}</h2>
      <button onClick={() => setCount(count + 1)}>Re-render</button>
      <p>Count: {count}</p>
    </div>
  );
}
```

Result: - `sum` is calculated once - Recalculated only if `numbers` changes

Example 2 — Filtering Large List

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

export default function UserList({ users }) {
  const [search, setSearch] = useState("");

  const filteredUsers = useMemo(() => {
    return users.filter(u =>
u.name.toLowerCase().includes(search.toLowerCase()));
  }, [users, search]);

  return (
    <div>
      <input value={search} onChange={e => setSearch(e.target.value)} />
      <ul>
        {filteredUsers.map(u => <li key={u.id}>{u.name}</li>)}
      </ul>
    </div>
  );
}
```

---

5. useCallback — Function Memoization

Definition (Interview Ready)

- Returns a stable function reference
- Prevents recreation of functions every render
- Works best with React.memo children
- Controlled by dependency array

Syntax

```
const memoizedFn = useCallback(() => {
  doSomething();
}, [dependency]);
```

Working Example

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

const Child = React.memo(({ item, removeItem }) => {
  console.log("Child rendered: ", item);
  return <li onClick={() => removeItem(item)}>{item}</li>;
});

export default function ItemList() {
  const [items, setItems] = useState(["A", "B", "C"]);

  const removeItem = useCallback((item) => {
    setItems(prev => prev.filter(i => i !== item));
  }, []);

  return (
    <ul>
      {items.map(item => (
        <Child key={item} item={item} removeItem={removeItem} />
      ))}
    </ul>
  );
}
```

Why this matters: - `removeItem` keeps the same reference - Child components do NOT re-render unnecessarily

---

6. Differences (Quick Comparison)

Feature	React.memo	useMemo	useCallback
Memoizes	Component	Value	Function
Purpose	Prevent re-render	Avoid recomputation	Avoid new functions
Works on	Props	Expensive logic	Callbacks
Return	Component	Computed value	Function

---

7. When to Use vs Not Use

Use React.memo when:

- Component is large
- Renders frequently
- Props rarely change

Avoid React.memo when:

- Component is tiny
- Props change every render

Use useMemo when:

- Sorting large arrays
- Filtering big lists
- Heavy mathematical operations

Use useCallback when:

- Passing functions to memoized children
- Handlers trigger minimal logic

---

8. Interview-Level Questions (With Answers)

1. What is the main difference between useMemo and useCallback?

Answer: - **useMemo** memoizes a **value (result of a computation).** - **useCallback** memoizes a **function reference.** - useMemo returns a value; useCallback returns a function.

---

2. Can React.memo replace useMemo?

Answer: No. They serve different purposes. - **React.memo** prevents component re-renders. - **useMemo** prevents expensive calculations inside a component. You may use both together in real applications.

---

3. Does React.memo guarantee no re-render?

Answer: No. A component wrapped with React.memo will still re-render if: - Its props change (shallow comparison fails), or - Its parent re-renders and passes new object/function props, or - It uses its own state or context that changes.

---

4. Why is shallow comparison important?

Answer: React.memo checks props using shallow comparison: - Primitive values (number, string, boolean) are compared by value. - Objects and arrays are compared by reference, not by content. If a new object is created on every render, React.memo becomes ineffective.

---

5. When does memoization hurt performance?

Answer: Memoization can hurt performance when: - The computation is very small or cheap. - Dependencies change frequently. - Too many memoized values increase memory usage. - Overusing memoization adds unnecessary complexity.

---

6. How does dependency array affect memoization?

Answer: - If dependencies **do not change**, React reuses the memoized value/function. - If any dependency **changes**, React recomputes. - Missing dependencies can cause stale (incorrect) data. - Extra dependencies can reduce optimization benefits.

---

7. Explain a real project use case of all three.

Answer (Dashboard Example): - **React.memo** → Memoize Chart and Table components. - **useMemo** → Memoize sorted and filtered analytics data. - **useCallback** → Memoize event handlers passed to child widgets.

---

9. Golden Rule to Remember

React.memo = Component useMemo = Value useCallback = Function