Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)

Question 1: What are React hooks? How do useState() and useEffect() work in functional components?

Ans : Hooks are special functions in React that let you use state and lifecycle features in functional components (before hooks, only class components had these features).

llows functional components to have **state**.

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


function Counter() {
  const [count, setCount] = useState(0); // state variable


  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

useEffect()

- Used for **side effects** (e.g., fetching data, subscriptions, updating DOM).
- Runs **after render**.

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

```
function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => setSeconds(s => s + 1), 1000);
    return () => clearInterval(interval); // cleanup
  }, []); // empty dependency = run once on mount

  return <p>Timer: {seconds}s</p>;
}
```

Question 2: What problems did hooks solve in React development? Why are hooks important?

Ans : Before hooks:

- You had to use class components for state & lifecycle.
- Logic was scattered (e.g., componentDidMount, componentDidUpdate had related code far apart).
- Code reuse was difficult (needed Higher-Order Components or Render Props).

Hooks solved this:

- Now you can use state & lifecycle in functional components.
- Make logic reusable and cleaner (custom hooks).
- Avoids "wrapper hell" from too many HOCs.

- Functional components + hooks are simpler and more powerful.

Question 3: What is useReducer? How do we use it?

Ans : useReducer is an alternative to useState.

Best for complex state logic (multiple related values or advanced updates).

Works like Redux: state + reducer function + actions.

```
import { useReducer } from "react";

function reducer(state, action) {
  switch (action.type) {
    case "increment": return { count: state.count + 1 };
    case "decrement": return { count: state.count - 1 };
    default: return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: "increment"
})}>+</button>
      <button onClick={() => dispatch({ type: "decrement" })}>-
</button>
    </div>
  );
}
```

Question 4: What is the purpose of useCallback & useMemo?

Ans : Both are performance optimization hooks.

- useCallback → Caches a function so it's not re-created on every render.
- useMemo → Caches a computed value so expensive calculations don't re-run unnecessarily.

Question 5: Difference between useCallback & useMemo

Ans :

Hook	**Returns**	**Used for**
useCallback(fn, deps)	Memoized function	Avoid re-creating functions unnecessarily
useMemo(fn, deps)	Memoized value	Avoid re-calculating expensive values

Example:

```
const memoizedValue = useMemo(() =>
expensiveCalculation(num), [num]);


const memoizedCallback = useCallback(() => {
  console.log("Called with", num);
}, [num]);
```

Question 6: What is useRef? How does it work?

- **Ans :** useRef returns a mutable object with .current property.
- Unlike state, updating .current does not re-render the component.

Uses:

1. Access DOM elements directly.
2. Store values between renders (like instance variables).

Example 1: DOM reference

```
import { useRef, useEffect } from "react";


function InputFocus() {
  const inputRef = useRef(null);


  useEffect(() => {
    inputRef.current.focus(); // focus input on mount
  }, []);

  return <input ref={inputRef} type="text" placeholder="Auto
focused!" />;
}
```

Example 2: Storing values without re-render

```
function Timer() {
  const countRef = useRef(0);

  const increment = () => {
    countRef.current += 1;
    console.log("Count:", countRef.current); // updates but no re-
render
  };

  return <button onClick={increment}>Increase</button>;
}
```