Beginner Questions (1-10)

1. What is the purpose of the `key` prop in React lists?

Answer: The `key` prop helps React identify which items have changed, been added, or removed, enabling efficient re-rendering of lists.

Explanation: Keys should be unique among siblings (but don't need to be globally unique). Using indexes as keys is discouraged if the list order may change.

---

2. How do you conditionally render components in React?

Answer: Using JavaScript operators like `&&` or ternary operators:

```
{isLoggedIn && <UserProfile />}
{isLoading ? <Spinner /> : <Content />}
```

---

3. What's the difference between props and state?

Answer: Props are read-only data passed from parent to child, while state is mutable data managed within the component.

Explanation: Props are for configuration, state is for interactivity. Changing props re-renders the child component, changing state re-renders the current component.

---

4. How do you create a controlled component?

Answer: By tying the input value to state and updating it via `onChange`:

```
function Input() {
  const [value, setValue] = useState('');
```

```
  return <input value={value} onChange={(e) => setValue(e.target.value)}
}
```

---

5. What does `setState` do in class components?

Answer: It schedules an update to the component's state object and triggers a re-render.

Explanation: `setState` is asynchronous - multiple calls may be batched. You can pass a function to access previous state: `setState(prev => prev + 1)`.

---

6. What is the purpose of `React.Fragment`?

Answer: It lets you group elements without adding extra DOM nodes.

Explanation: Helpful when a component needs to return multiple adjacent elements. Short syntax: `<>...</>`.

---

7. How do you handle events in React?

Answer: With camelCase event handlers like `onClick`:

```
<button onClick={() => console.log('Clicked')}>Click</button>
```

8.What are React hooks?

Answer: Functions that let you use state and other React features in function components.

Explanation: Examples include `useState`, `useEffect`, `useContext`. Hooks must be called at the top level (not in loops/conditions).

---

9. How do you pass data from child to parent component?

Answer: By passing callback functions as props:

```
function Parent() {
  const handleChildEvent = (data) => console.log(data);
  return <Child onEvent={handleChildEvent} />;
}
```

---

10. What is prop drilling?

Answer: The process of passing props through multiple levels of components.

Explanation: This can become cumbersome in deep component trees, solved by Context API or state management libraries.

Intermediate Questions (11-20)

Explain the component lifecycle in class components.

Answer: Key lifecycle methods:

- **Mounting:** `constructor`, `render`, `componentDidMount`
- **Updating:** `shouldComponentUpdate`, `render`, `componentDidUpdate`
- **Unmounting:** `componentWillUnmount`

What are Higher-Order Components (HOC)?

Answer: Functions that take a component and return a new enhanced component.

Example:

```
const withAuth = (Component) => (props) =>
  isAuthenticated ? <Component {...props} /> : <Login />;
```

How does `useMemo` optimize performance?

Answer: It memoizes expensive calculations:

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

What are React Portals?

Answer: A way to render children outside the DOM hierarchy:

```
ReactDOM.createPortal(child, domNode)
```

Use case: Modals, tooltips that need to break out of container CSS.

Explain the rules of hooks.

Answer:

1. Only call hooks at the top level (not in loops/conditions)
2. Only call hooks from React functions (components or custom hooks)

What is the difference between `useEffect` and `useLayoutEffect`?

Answer: `useLayoutEffect` fires synchronously after DOM mutations but before paint, while `useEffect` fires asynchronously after render and paint.

How do you prevent unnecessary re-renders?

Answer: Using:

- `React.memo` for components
- `useMemo` for values
- `useCallback` for functions
- `shouldComponentUpdate` in class components

What is code splitting in React?

Answer: Splitting code into smaller bundles loaded on demand:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

How do you handle forms with multiple inputs?

Answer: Using a single state object and computed property names:

```
const [form, setForm] = useState({name: '', email: ''});
const handleChange = (e) => {
  setForm({...form, [e.target.name]: e.target.value});
};
```

What are error boundaries?

Answer: Class components that catch JavaScript errors in their child component tree.

Implementation:

```
class ErrorBoundary extends React.Component {
  state = { hasError: false };
  static getDerivedStateFromError() { return { hasError: true }; }
  componentDidCatch(error, info) { logError(error, info); }
  render() { return this.state.hasError ? <Fallback /> : this.props.childc
}
```

Advanced Questions (21-30)

Explain how React's reconciliation algorithm works.

Answer: React's reconciliation algorithm is responsible for determining the most efficient way to update the DOM when a component's state or props change. When the component renders, React builds

a new "virtual DOM," which is a lightweight representation of the actual DOM. React then compares the new virtual DOM with the previous version (this process is called "diffing") and calculates the minimal number of changes required to update the real DOM. This ensures that React only makes necessary updates, improving performance.

Key points:

Heuristic O(n) algorithm: React uses a heuristic approach with an O(n) time complexity to efficiently update the DOM, even for large applications.

Element comparison and keys: React compares the element types (e.g., div vs. span) to check if they have changed. For lists, React uses key attributes to uniquely identify list elements, which helps optimize updates by reusing components if their keys haven't changed.

How would you implement a custom `usePrevious` hook?

Answer: A usePrevious hook helps store the previous value of a state or prop between renders. React doesn't provide this functionality out of the box, but it can be implemented using a useRef to store the previous value and useEffect to update it when the value changes.

```
function usePrevious(value) {
  const ref = useRef(); // Create a mutable reference
  useEffect(() => {
    ref.current = value; // Update the reference with the current value
  }, [value]); // Run this effect whenever 'value' changes
  return ref.current; // Return the previous value
}
```

This hook is useful when you need to compare the previous and current values of a prop or state, such as when tracking changes in an input field or comparing the current and previous states.

What are compound components?

Answer: Compound components are a pattern in React where a parent component provides a set of child components that work together and share an implicit state. This allows components to be more flexible and reusable without relying on props passing down explicitly.

```
<Toggle>
  <Toggle.On>The button is on</Toggle.On>
  <Toggle.Off>The button is off</Toggle.Off>
  <Toggle.Button />
</Toggle>
```

In this example:

Toggle is the parent component, and it holds the shared state (whether the button is on or off).

Toggle.On, Toggle.Off, and Toggle.Button are children that are implicitly aware of the parent component's state. The parent manages the state and passes it down to the children components without the need for props.

This pattern makes it easy to create components like toggles, modals, or tabs that have interconnected behavior.

Implement a debounce hook.

Answer: Debouncing is a technique to limit the number of times a function is called, especially for expensive operations like API calls or user input handling. The useDebounce hook delays the update until after the user stops typing or performing the action for a specified delay period.

```
function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);
  useEffect(() => {
    const timer = setTimeout(() => setDebouncedValue(value), delay);
    return () => clearTimeout(timer); // Cleanup the timer when the value
  }, [value, delay]); // Re-run the effect when 'value' or 'delay' change
  return debouncedValue;
}
```

In this hook:

1. The debouncedValue state is updated after the value has stopped changing for a specified delay.
2. This is useful for preventing unnecessary function calls, especially in situations like form input fields, where you don't want to trigger an API call every time the user types.

What is the Context API and when should you use it?

Answer: The Context API is a feature in React that allows you to share data (such as theme settings, authentication, or language preference) across components without having to manually pass down props at every level. It is particularly useful when you need to share state across many components at different levels of the component tree.

```
const ThemeContext = createContext();
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Child />
    </ThemeContext.Provider>
  );
}

function Child() {
  const theme = useContext(ThemeContext);
  return <div>{theme}</div>;
}
```

In this example:

1. ThemeContext is created with createContext(), and the value is provided at the top level of the app using ThemeContext.Provider.
2. Any child component can access the shared theme value using useContext(ThemeContext).
3. Context is useful for state that needs to be accessed by many components, like user authentication status or theme preferences.

How does React's concurrent mode work?

Answer: React's Concurrent Mode is an experimental feature that allows React to work on multiple updates concurrently. Unlike the traditional React rendering model, which updates the DOM in a single process, Concurrent Mode enables React to break up rendering work into smaller chunks and prioritize more important updates (such as animations or user interactions). This helps keep the app responsive and smooth, even under heavy workloads.

Features of Concurrent Mode:

1. Time slicing: Divides rendering work into small chunks to allow the browser to do other tasks, ensuring that the app remains responsive.

2. Suspense: Helps to manage asynchronous rendering by "suspending" parts of the app while waiting for data (e.g., loading a component or fetching data).
3. Transition updates: Lets you mark certain updates as less urgent, so React can prioritize important ones, like user interactions.

How would you optimize a large list rendering ?

Answer: Rendering a large list of items can cause performance issues because React will attempt to render every item, even if it's not visible on the screen. To optimize this, you can use windowing or virtualization, which means only rendering the items that are visible in the viewport and reusing those components as you scroll.

For example, using the `react-window` library:

Features of Concurrent Mode:

```
<List height={600} itemCount={1000} itemSize={35}>
  {Row}
</List>
```

Here:

1. The List component only renders items that fit in the height of the list container.
2. As you scroll, it dynamically renders new items, improving performance.

Explain how to test React components ?

Answer: Testing React components ensures your code works as expected. To test React components, you can use the following tools:

Jest: A JavaScript test runner used to run tests and assert conditions.

React Testing Library: A library that helps you test components by rendering them in a way that simulates how users interact with them (via the DOM).

Mocking: You can use jest.mock to mock dependencies and isolate components for testing.

Testing hooks: Use @testing-library/react-hooks to test custom hooks.

```
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';
```

```
test('renders the correct text', () => {
  render(<MyComponent />);
  expect(screen.getByText('Hello World')).toBeInTheDocument();
});
```

How does React's new Server Components differ from SSR?

Answer: React Server Components and Server-Side Rendering (SSR) both allow for rendering content on the server, but they have key differences:

Server Components:

1. Rendered entirely on the server, meaning they don't contribute to the JavaScript bundle size.
2. Can access backend resources directly and fetch data without affecting the client-side bundle.
3. Support progressive enhancement, where only certain parts of the page are enhanced on the client-side after initial load.
4. Complement SSR, allowing for better server-side integration without overloading the client-side bundle.

SSR:

1. Renders the entire component on the server and sends HTML to the client, often requiring a full page reload or hydration of the client-side application.
2. Server Components are more lightweight and efficient in certain scenarios, as they avoid sending unnecessary JavaScript to the client.

Input/Output Questions (31-40)

31. What does this code output?

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log(count);
  }, [count]);
```

```
  return <button onClick={() => setCount(c => c + 1)}>Click</button>;
}
```

Input/Output Questions (31-40)

31. What does this code output?

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log(count);
  }, [count]);
  return <button onClick={() => setCount(c => c + 1)}>Click</button>;
}
```

Output: Logs `0` on mount, then logs incremented value on each click.

Explanation:
The useEffect hook is triggered on mount (initial render) and on every update to the count state. When the component mounts, count is 0, and the effect logs it. After each button click, the setCount function updates the count, and the effect logs the updated value. The update is asynchronous, so the effect will log the updated value after each render.

---

32. What's the output of this component?

```
function MyComponent() {
  console.log('render');
  return <div>Hello</div>;
}
```

Output: "render" logged once (unless parent re-renders it).

Explanation:
The component simply logs "render" to the console each time it renders. Since there are no state updates or props changing in this component, it will only log the message once during the initial render.

If the parent component re-renders, this component will also re-render, and "render" will be logged again.

---

33. What happens when this component renders?

```
function Counter() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1);
    }, 1000);
    return () => clearInterval(id);
  }, []);
  return <div>{count}</div>;
}
```

Output: Count increments to `1` then stops (stale closure issue).

Fix: Use `setCount(c => c + 1)` .

Explanation:

The useEffect hook sets up an interval that increments the count every second. However, the setCount function uses the count value from the initial render (due to a stale closure). This means it will only increment to 1 and stop. The fix is to use the functional update form of setCount, which will always receive the latest value of count.

---

34. What's the order of console logs?

```
class Example extends React.Component {
  componentDidMount() { console.log('mount'); }
  componentDidUpdate() { console.log('update'); }
  render() { console.log('render'); return null; }
}
```

Output: "render", "mount" on initial render.
Then "render", "update" on updates.

Explanation:

When the component first mounts, the render method is called, followed by componentDidMount. After

any subsequent updates to the component, the render method will be called again, followed by componentDidUpdate. The order of the logs is determined by the lifecycle methods in React.

---

35. What does this code display?

```
const Context = React.createContext();
function App() {
  return (
    <Context.Provider value="A">
      <Context.Provider value="B">
        <Child />
      </Context.Provider>
    </Context.Provider>
  );
}
function Child() {
  const value = useContext(Context);
  return <div>{value}</div>;
}
```

Output: "B" (nearest provider value is used).

Explanation:
React's useContext hook allows a component to consume values provided by a Context.Provider. In this case, the nearest Context.Provider around the Child component provides the value "B", so that is what gets rendered. Even though there is another Context.Provider higher up the tree with the value "A", the nearest Provider takes precedence.

---

36. What's the output after clicking?

```
function App() {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1);
    setCount(count + 1);
  };
  return <button onClick={handleClick}>{count}</button>;
}
```

Output: Count increments by `1` (state updates are batched).

Explanation:
In React, state updates within the same event handler are batched together for performance. So, both setCount(count + 1) calls are executed together, but React uses the initial count value for both calls. As a result, the state is updated only once, and the displayed count is incremented by 1 instead of 2.

---

37. What does this custom hook output?

```
function useCounter(initial) {
  const [count, setCount] = useState(initial);
  const increment = useCallback(() => setCount(c => c + 1), []);
  return [count, increment];
}
```

Output: Returns current count and stable increment function.

Explanation:
This custom hook returns the count and an increment function that can be used to increase the count. The useCallback hook ensures that the increment function is stable and doesn't change unless its dependencies (which are empty in this case) change. This prevents unnecessary re-renders of components that use this hook.

---

38. What's the render output and is there any problem with the following code?

```
function List({ items }) {
  return (
    <ul>
      {items.map((item, i) => <li key={i}>{item}</li>)}
    </ul>
  );
}
```

Output: Renders list but with potential issues if items reorder (index keys).

Explanation:
The key prop in React helps optimize re-renders by uniquely identifying elements in a list. However,

using the index (i) as the key can cause issues when the list is reordered because React might not correctly associate items with their previous state. It is better to use a unique identifier as the key if possible.

---

39. What happens when this runs?

```
function EffectDemo() {
  const [data, setData] = useState(null);
  useEffect(() => {
    async function fetchDataAsync() {
      const result = await fetchData();
      setData(result);
    }
    fetchDataAsync();
  }, []);
  return <div>{data}</div>;
}
```

Output: Warning (effect callback shouldn't be `async`).

Fix: Define `async` function inside the effect.

Explanation:
React's useEffect callback cannot be an async function directly. This is because async functions return a Promise, and React expects the callback to return either undefined or a cleanup function. The solution is to define an async function inside the useEffect and call it, as shown in the fixed version.

---

40. What's the final output?

```
function Example() {
  const [value, setValue] = useState(0);
  useEffect(() => {
    if (value === 0) {
      setValue(10);
    }
  }, [value]);
  return <div>{value}</div>;
}
```

Output: Initially `0`, then `10` (effect triggers state update).

Explanation:
The useEffect hook runs after every render, and it checks if value is 0. Initially, value is 0, so setValue(10) is called, causing a re-render. On the second render, value is 10, and the effect no longer updates the state since the condition value === 0 is not met.