Top React Interview Questions (2025)

September 17, 2025 — UNCATEGORIZED No Comments 79 Mins Read — By akshay

This guide is a carefully curated collection of **75 React interview questions** designed to help learners build a deep understanding of React and confidently crack technical interviews at product-based companies, startups, and top-tier organizations.

The questions are structured progressively, starting from the fundamentals and moving toward advanced concepts, including state management, hooks, performance optimization, server-side rendering, error handling, and more. Each question is accompanied by:

1. The full question text.
2. The topic it belongs to.
3. Difficulty level (Easy / Intermediate / Advanced).
4. A detailed explanation with code examples where applicable.
5. Common interview scenarios where the question might arise.
6. Additional best practices, pitfalls, and optimization tips where relevant.

Whether you're preparing for your first React interview or aiming for a senior frontend role, this guide covers everything you need to confidently discuss React concepts, debug problems, and architect scalable solutions.

How to use this guide:

- Study each question thoroughly.
- Write and practice coding examples yourself.

- Revise the best practices and pitfalls.
- Mock interviews using the sample scenarios provided.
- Reinforce learning by building small projects using the concepts covered.

With consistent practice and understanding, you'll be well-equipped to answer React-related interview questions with clarity, confidence, and depth.

Let's begin this journey toward mastering React!

---

1. What is React? Describe the benefits of React | React Basics | Easy

React is a JavaScript library created by Facebook for building user interfaces, especially single-page applications (SPAs).

It helps developers build reusable UI components that can manage their own state, which makes it easier to build complex applications.

Key features include:

- **Component-based architecture:** Developers can break the UI into small, reusable components.
- **Virtual DOM:** React updates the UI efficiently by calculating differences between the new and old DOM trees.
- **Declarative UI:** You describe how the UI should look for any given state, and React manages the rendering.
- **One-way data binding:** Data flows from parent to child components, making state management predictable.
- **JSX syntax:** Allows mixing HTML and JavaScript, making component structure more readable.

Example:

```
import React from 'react';

function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}

function App() {
  return <Welcome name="Alice" />;
}

export default App;
```

In this example, the `Welcome` component takes a `name` prop and renders it dynamically.

Interview Scenarios:

- Explaining React's role in a web application during a technical round.
- Answering why React is preferred over other libraries/frameworks for SPA development.
- Discussing how reusable components can help scale applications.
- Talking about performance optimizations using Virtual DOM in a system design interview.

---

2. What is the difference between React Node, React Element, and a React Component? | React Core Concepts | Easy

In React:

- A **React Node** is anything that can be rendered, such as a React element, string, number, array, or `null`.

- A **React Element** is an immutable object that describes what should appear in the UI. It's created using JSX or `React.createElement()`.

- A **React Component** is a function or class that returns a React element and encapsulates logic and UI behavior.

Example:

```
// React Element
const element = <h1>Hello, world!</h1>;

// React Component
function Greeting() {
  return <h1>Hello!</h1>;
}

// Using the component
const node = <Greeting />;
```

Interview Scenarios:

- Clarifying the difference between various React constructs during an entry-level interview.

- Asking how React builds and renders components efficiently.

- Discussing the importance of immutability and declarative rendering.

---

3. What is JSX and how does it work? | React Syntax | Easy

JSX (JavaScript XML) is a syntax extension that allows writing HTML-like code inside JavaScript. It makes defining UI structures more intuitive.

JSX is not understood by browsers directly, so tools like Babel transpile it into `React.createElement()` calls.

Example:

```
const element = <div>Hello, world!</div>;
```

Is transpiled to:

```
const element = React.createElement('div', null, 'Hello, world!');
```

Interview Scenarios:

- Asking how JSX simplifies component development.
- Exploring how transpilers work with React code.
- Understanding how React internally manages element creation.

---

4. What is the difference between state and props in React? | State Management | Easy

- **State:** Internal data managed by the component, can change over time.
- **Props:** External data passed from parent components, read-only within the component.

Example:

```
function Counter() {
  const [count, setCount] = React.useState(0); // state
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</buttor
    </div>
  );
}

function App() {
  return <Counter initialCount={0} />; // props can be passed here
}
```

Interview Scenarios:

- Asking how data flows in a React component hierarchy.
- Explaining why state and props serve different roles.
- Understanding how changes in state affect rendering.

---

5. What is the purpose of the `key` prop in React? | Lists & Rendering | Easy

The `key` prop helps React uniquely identify elements in a list. It optimizes rendering by tracking which items changed, were added, or removed.

Example:

```
const items = [
  { id: 1, value: 'Apple' },
  { id: 2, value: 'Banana' }
];
```

```
function ItemList() {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>{item.value}</li>
      ))}
    </ul>
  );
}
```

Interview Scenarios:

- Asking how React improves performance when rendering lists.
- Discussing how improper key usage can cause bugs.

---

6. What is the consequence of using array indices as the value for `key` in React? | Rendering / Lists | Intermediate

Using array indices as keys in React lists can lead to bugs and performance issues because React uses keys to identify which items have changed, been added, or removed.

- Keys should be stable, predictable, and unique.
- If array indices are used as keys, reordering, adding, or removing items can confuse React during reconciliation.
- This can cause components to unnecessarily re-render or preserve incorrect state.

Example:

```
const items = ['Apple', 'Banana', 'Cherry'];

items.map((item, index) => <div key={index}>{item}</div>);
```

If items are reordered, React may reuse DOM nodes incorrectly, leading to UI inconsistencies.

Best practices:

- Use unique identifiers (like IDs) for keys.
- Avoid using indices unless the list is static and won't change order.

Interview scenarios:

- Asked to explain list rendering and key importance.
- Debugging a component that behaves incorrectly after array reordering.

---

7. What is the difference between controlled and uncontrolled React Components? | Forms / State | Intermediate

Controlled components are form inputs whose values are managed by React state. Uncontrolled components maintain their own internal state and are accessed via refs.

Explanation:

- **Controlled:** The state is the single source of truth. Input value is tied to `state` and updated via `onChange`.
- **Uncontrolled:** Input value is stored in the DOM. Accessed using `ref` when needed.

Example (Controlled):

```
function ControlledInput() {
  const [value, setValue] = React.useState('');
```

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

Example (Uncontrolled):

```
function UncontrolledInput() {
  const inputRef = React.useRef();

  const handleClick = () => {
    alert(inputRef.current.value);
  };

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>Show Value</button>
    </>
  );
}
```

Best practices:

- Use controlled components for form validation, complex forms, or dynamic updates.
- Use uncontrolled components for simple forms or when integrating with third-party libraries.

Interview scenarios:

- Asked to explain differences between controlled vs uncontrolled components.
- Asked when to use `ref` for form inputs.

---

8. What are some pitfalls about using context in React? | State Management / Context | Intermediate

React Context provides a way to share global state, but improper usage can lead to performance and maintainability issues.

Explanation:

- Context updates propagate to all consuming components.
- Overuse can lead to unnecessary re-renders.
- Using large or frequently changing data in context may degrade performance.

Common pitfalls:

- Putting too much state in a single context.
- Passing non-memoized objects as context values.
- Relying solely on context instead of external state management for complex apps.

Best practices:

- Memoize context values using `useMemo`.
- Split contexts to isolate frequently changing state.
- Use libraries like Redux or Zustand for complex scenarios.

Interview scenarios:

- Asked about performance issues with context.
- Asked to optimize a component consuming context for frequently updating data.

---

9. What are the benefits of using hooks in React? | Hooks / Functional Components | Beginner

Hooks allow functional components to use state and other React features without writing classes.

Explanation:

- **Simpler code:** Eliminates the need for class components and lifecycle methods.
- **Reusable logic:** Custom hooks allow encapsulating stateful logic.
- **Better readability:** Clear separation of state and effects.
- **Functional approach:** Encourages a functional programming style, making code easier to test and maintain.

Example:


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

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

  useEffect(() => {
    console.log(`Count is: ${count}`);
  }, [count]);

  return <button onClick={() => setCount(count + 1)}>Increment</but
}
```

Best practices:

- Use hooks only in functional components.
- Follow rules of hooks (top-level calls, no conditional calls).
- Extract reusable logic into custom hooks.

Interview scenarios:

- Asked why hooks replaced class components.

- Asked to refactor class component into functional component using hooks.

---

10. What are the rules of React hooks? | Hooks / Functional Components | Intermediate

Hooks have rules to ensure predictable behavior and maintain component integrity.

Explanation:

1. **Call hooks at the top level:** Don't call inside loops, conditions, or nested functions.
2. **Call hooks only in React functions:** Functional components or custom hooks.
3. **Custom hooks must start with "use":** Helps React identify hook calls.

Why rules matter:

- Maintaining the order of hook calls ensures React can correctly preserve state across renders.
- Violating rules can lead to unpredictable behavior or errors.

Example (Incorrect use):

```
function MyComponent() {
  if (someCondition) {
    const [value, setValue] = useState(0); // ❌ called conditiona
  }
}
```

Correct use:

```
function MyComponent() {
  const [value, setValue] = useState(0); //always called
```

```
    if (someCondition) {
      // logic
    }
  }
```

Interview scenarios:

- Asked to explain why hooks must be top-level.
- Asked to identify why a component using hooks fails unexpectedly.

---

11. What is the difference between `useEffect` and `useLayoutEffect` in React? | Hooks – Side Effects | Intermediate

`useEffect` and `useLayoutEffect` are two hooks in React used to perform side effects, such as data fetching, subscriptions, or DOM manipulations. Although they are similar in API, they differ in when they execute during the component lifecycle.

Key Differences:

Aspect	`useEffect`	`useLayoutEffect`
Execution timing	Runs after the browser has painted the UI	Runs after DOM updates but before paint
Suitable for	Data fetching, subscriptions, event listeners	Measuring or synchronizing DOM elements before user sees it
Blocking behavior	Doesn't block rendering	Blocks rendering until executed

Example:

```
import React, { useState, useEffect, useLayoutEffect, useRef } from

function Example() {
  const [width, setWidth] = useState(0);
  const divRef = useRef();

  useLayoutEffect(() => {
    // Runs before paint, ideal for measuring layout
    setWidth(divRef.current.offsetWidth);
  }, []);

  useEffect(() => {
    console.log("Rendered with width:", width);
  }, [width]);

  return <div ref={divRef} style={{ width: '50%' }}>Resize me!</div
}
```


When to use each:

- **`useEffect`:**

- Fetching data after initial render
- Adding event listeners or subscriptions
- Performing actions that don't affect layout calculations

- **`useLayoutEffect`:**

- Measuring DOM elements before paint
- Synchronously updating DOM styles
- Adjusting scroll position or animations before user interaction

Mistakes to avoid:

- Using `useLayoutEffect` for data fetching unnecessarily blocks rendering, causing a poor user experience.
- Forgetting to clean up effects like event listeners, which can cause memory leaks.
- Assuming both hooks behave the same – `useLayoutEffect` blocks paint, so it should be used carefully.

Best practices:

- Default to using `useEffect` unless layout measurements or synchronous DOM updates are required.
- Always clean up side effects in the return function to avoid resource leaks.

```
useEffect(() => {
  const handleResize = () => { /* logic */ };
  window.addEventListener('resize', handleResize);
  return () => window.removeEventListener('resize', handleResize);
}, []);
```

Interview scenarios:

- You are asked to explain how React manages updates and why `useLayoutEffect` might be necessary in animation-heavy applications.
- You are given a component that flickers when updated and asked to debug why the layout measurement might be happening too late.
- You are asked how to manage side effects in components and what the difference between synchronous and asynchronous updates is.

---

12. What is the purpose of callback function argument format of `setState()` in React and when should it be used? | State Management | Intermediate

The callback function format of `setState()` ensures that state updates are calculated based on the most recent state and props. It's important when the new state depends on the current state because React batches updates asynchronously.

Explanation:

React's `setState()` doesn't immediately update the state instead, it schedules an update. If you calculate the next state based on the current state directly, you may end up using a stale value.

The callback format takes two arguments:

- `prevState` : The state before the update
- `props` : The latest props passed to the component

This ensures your update logic always references the correct state.

Example:

```
class Counter extends React.Component {
  state = { count: 0 };

  increment = () => {
    this.setState((prevState, props) => ({
      count: prevState.count + 1
    }));
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
```

```
        </div>
      );
    }
  }
```

If you used this instead, it could cause bugs:

```
this.setState({
  count: this.state.count + 1
});
```


Because multiple calls to `increment` might not correctly use the updated value of `count`.

When should it be used:

- When updating state based on previous state, such as toggling values, counters, or accumulative calculations.
- When updates need to ensure they are using the latest state, especially in event handlers or asynchronous operations.

Mistakes to avoid:

- Updating state directly using `this.state` instead of `prevState` may lead to inconsistent updates.
- Not recognizing that state updates are batched and asynchronous, especially in concurrent rendering.

Best practices:

- Always use the callback format when the new state depends on previous values.
- Avoid side effects within `setState()` callbacks keep logic pure and predictable.

Interview scenarios:

- Asked to implement a counter where multiple increments happen quickly.
- Asked to explain how React manages asynchronous state updates.
- Asked why certain bugs happen when state is updated based on stale values.

---

13. What does the dependency array of `useEffect` affect? | Hooks Side Effects | Intermediate

The dependency array determines when the effect should be re-executed after renders. It's crucial to manage this correctly to avoid unnecessary reruns or stale data.

Explanation:

- If the dependency array is empty ([ ]), the effect runs once after the initial render.
- If the array includes variables, the effect runs whenever one of them changes.
- If omitted, the effect runs after every render, which can lead to performance issues.

Example:

```
import React, { useState, useEffect } from 'react';

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

  useEffect(() => {
    console.log('Effect runs');
    const interval = setInterval(() => {
      setCount(c => c + 1);
    }, 1000);

    return () => clearInterval(interval);
  }, []); // Empty array: effect runs once on mount
```

```
    return <div>Count: {count}</div>;
  }
```

If we used `[count]`, it would re-run the effect on every count update, unnecessarily creating new intervals.

Mistakes to avoid:

- Forgetting to include all relevant dependencies, which can cause bugs due to stale closures.
- Adding too many dependencies, causing unnecessary rerenders.
- Omitting the array entirely, which makes effects run on every render.

Best practices:

- Include all dependencies that are referenced inside the effect to avoid stale data.
- Use `useCallback` or `useMemo` to stabilize functions or values used as dependencies.
- Use ESLint's `react-hooks/exhaustive-deps` rule to catch missing dependencies.

Interview scenarios:

- Asked how effects can cause bugs due to stale closures.
- Asked how to handle API calls inside `useEffect`.
- Asked why developers should not omit the dependency array.

---

14. What is the `useRef` hook in React and when should it be used? | Hooks Refs | Intermediate

`useRef` creates a persistent reference that doesn't cause rerenders when updated. It's commonly used to access DOM elements or keep mutable state across renders.

Explanation:

- Returns a mutable object `{ current: ... }`.

- Persists across renders, making it suitable for storing values without triggering updates.
- Useful for:
- Accessing DOM nodes
- Storing previous state
- Holding intervals or timers
- Managing focus

Example (Focus input):


```
import React, { useRef, useEffect } from 'react';

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

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} type="text" />;
}
```

When to use:

- Manipulating DOM directly (focus, scroll, measurements).
- Storing mutable values that should not cause rerenders.
- Keeping references to timers, WebSocket connections, or subscriptions.

Mistakes to avoid:

- Using `useRef` for state that should trigger UI updates use `useState` instead.
- Forgetting that `useRef` does not cause rerenders when its value changes.
- Overusing refs instead of proper state management.

Best practices:

- Use `useRef` for non-render affecting data.
- Keep ref logic outside of component state to avoid unnecessary complexity.
- Avoid using it as a substitute for state management when state-driven UI changes are required.

Interview scenarios:

- Asked to manage form input focus.
- Asked how to store previous render values without triggering rerenders.
- Asked how to manipulate DOM directly within functional components.

---

15. What is the `useCallback` hook in React and when should it be used? | Hooks Performance Optimization | Intermediate

`useCallback` memoizes functions so that they are only recreated when dependencies change. It's a performance optimization that prevents unnecessary rerenders, especially in child components.

Explanation:

Functions in JavaScript are recreated every time a component rerenders. If a function is passed as a prop to a child, the child may rerender unnecessarily. `useCallback` ensures that the function reference stays stable unless its dependencies change.

Example:

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

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

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

  return <Child onIncrement={increment} />;
}

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


In this example, `Child` will not rerender unnecessarily because the `increment` function is memoized.

When to use:

- Passing callbacks to optimized child components.
- Avoiding function recreation when dependencies are unchanged.
- Managing expensive event handlers.

Mistakes to avoid:

- Using `useCallback` indiscriminately it adds overhead and complexity without real benefits in simpler cases.
- Forgetting to include dependencies, leading to stale references.
- Overusing memoization without profiling.

Best practices:

- Use `useCallback` when functions are passed as props and you want to prevent rerenders.
- Include all relevant dependencies in the dependency array.
- Profile performance before over-optimizing.

Interview scenarios:

- Asked how to avoid unnecessary renders in performance-critical applications.
- Asked to explain how closures and stale references occur in hooks.
- Asked why memoization is important in component hierarchies.

---

16. What is the `useMemo` hook in React and when should it be used? | Performance Optimization | Intermediate

`useMemo` is a React hook that memoizes the result of a function so that it is only recalculated when its dependencies change. This helps avoid expensive computations on every render and improves performance.

Explanation:

In React, computations or derived state that rely on props or state might be expensive. Without memoization, these computations would run every time the component rerenders. `useMemo` caches the result and recalculates only when dependencies change.

Example:

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

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

  const factorial = useMemo(() => {
    console.log('Calculating factorial...');
    return computeFactorial(count);
  }, [count]);

  return (
    <div>
      <h1>Factorial of {count} is {factorial}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</buttor
```

```
        <input value={text} onChange={e => setText(e.target.value)} p
      </div>
    );
  }

  function computeFactorial(n) {
    if (n <= 0) return 1;
    return n * computeFactorial(n - 1);
  }
```

In this example, changing `text` won't recalculate the factorial because the dependency array ensures it only recomputes when `count` changes.

When to use:

- Avoid expensive computations on every render.
- Memoize derived data from props or state.
- Improve performance in large component trees.

Mistakes to avoid:

- Overusing `useMemo` for trivial computations where caching overhead outweighs the benefits.
- Not specifying the correct dependencies, which can lead to stale or incorrect results.
- Assuming `useMemo` always improves performance without profiling.

Best practices:

- Use it when you have expensive computations.
- Always include all dependencies that affect the computation.
- Profile performance to justify memoization.

Interview scenarios:

- Asked how to optimize performance when components rerender frequently.
- Asked to explain how memoization avoids unnecessary computations.
- Asked to contrast `useMemo` and `useCallback`.

---

17. What is the `useReducer` hook in React and when should it be used? | State Management | Intermediate

`useReducer` is a hook that manages complex state logic by using a reducer function. It's an alternative to `useState`, suitable when state has multiple sub-values or when the next state depends on the previous state.

Explanation:

A reducer is a function that takes the current state and an action, and returns a new state. `useReducer` provides a more structured way of updating state, especially when state transitions are complex or require multiple branches of logic.

Example:

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch(action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error('Unknown action');
  }
}

function Counter() {
```

```
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: 'increment' })}>+</bu
      <button onClick={() => dispatch({ type: 'decrement' })}>-</bu
    </div>
  );
}

export default Counter;
```


When to use:

- When state is an object with multiple properties.
- When state updates depend on the current state.
- When managing forms, complex UI interactions, or nested state.

Mistakes to avoid:

- Using it when `useState` is sufficient, adding unnecessary complexity.
- Forgetting to handle unknown actions, which may cause errors.
- Not memoizing the reducer function when needed.

Best practices:

- Define clear action types.
- Separate state logic from component logic for readability.
- Use context with `useReducer` to share state across components.

Interview scenarios:

- Asked to manage complex form states with multiple inputs.
- Asked to implement undo/redo functionality using state history.

- Asked to refactor `useState` logic into `useReducer`.

---

18. What is the `useId` hook in React and when should it be used? | Accessibility / Forms | Intermediate

`useId` generates unique IDs that are stable across server and client renders. It is particularly useful for linking form elements, labels, and other accessibility-related attributes.

Explanation:

Generating unique IDs is important to prevent clashes when rendering forms or components multiple times. `useId` ensures IDs are unique even when server-side rendering (SSR) is used, which helps with hydration and accessibility.

Example:

```
import React, { useId } from 'react';

function LoginForm() {
  const id = useId();

  return (
    <form>
      <label htmlFor={id}>Username:</label>
      <input id={id} type="text" />
    </form>
  );
}
```

Here, the `id` generated by `useId` ensures that the `label` is correctly associated with the `input`.

When to use:

- Linking labels to form inputs.
- Generating unique IDs for accessibility attributes.
- Ensuring SSR compatibility with stable IDs.

Mistakes to avoid:

- Using random or incremental IDs that break during hydration.
- Forgetting to use `useId` when accessibility features require unique identifiers.
- Manually managing IDs in complex forms leading to duplication.

Best practices:

- Always use `useId` for form-related accessibility.
- Avoid global counters for generating IDs.
- Ensure IDs remain consistent across renders.

Interview scenarios:

- Asked to improve form accessibility.
- Asked how server-side rendering affects unique identifiers.
- Asked to explain how hydration issues arise in React.

---

19. What does re-rendering mean in React? | Rendering Concepts | Easy

Re-rendering refers to the process where React updates a component's output based on changes in its state or props. React recalculates the virtual DOM and updates the actual DOM to reflect those changes.

Explanation:

Whenever a component's state or props change, React schedules a rerender. It creates a new virtual DOM tree, compares it to the previous one using the reconciliation process, and applies only the differences to the real DOM.

Example:

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

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

  console.log("Component rendered");

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</buttor
    </div>
  );
}
```

Each time you click the button, the state changes, causing React to rerender the component.

When re-rendering happens:

- State updates using `useState` or `setState`.
- Props change from a parent component.
- Context value changes.
- Force updates triggered programmatically.

Mistakes to avoid:

- Causing unnecessary renders by not memoizing functions or objects passed as props.
- Deeply nesting state and causing complex rerenders.
- Forgetting that every state change triggers a rerender.

Best practices:

- Lift state only when necessary.
- Use `React.memo` or `useCallback` to optimize rerenders.
- Keep state flat and minimal to reduce complexity.

Interview scenarios:

- Asked how React handles updates efficiently.
- Asked to explain why components rerender after state changes.
- Asked to debug performance issues related to unnecessary rerenders.

20. What are React Fragments used for? | JSX / Rendering | Easy

React Fragments allow you to group multiple elements without adding extra nodes to the DOM. It's useful when you want to return multiple elements from a component without introducing unnecessary wrappers.

Explanation:

Normally, JSX requires a single parent element. React Fragments let you group elements without rendering extra DOM nodes, which can interfere with layout or styling.

Example:

```
import React from 'react';

function List() {
  return (
    <>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </>
```

```
    );
}
```

This way, the component returns multiple list items without wrapping them in a `<div>` or other container.

When to use:

- Grouping elements in a list without extra DOM nodes.
- Avoiding unnecessary wrappers that complicate styling or structure.
- Returning multiple elements from a component.

Mistakes to avoid:

- Forgetting to wrap elements when JSX expects a single parent.
- Using extra `<div>` containers that break CSS layouts.
- Overusing fragments where semantic HTML elements are better.

Best practices:

- Use fragments when grouping elements that don't need a container.
- Keep your component tree clean and avoid unnecessary DOM nodes.
- Use `React.Fragment` if you need to attach keys in lists.

Interview scenarios:

- Asked how to return multiple elements from a component.
- Asked why unnecessary DOM nodes can cause styling issues.
- Asked to explain differences between fragments and regular containers.

---

21. What is `forwardRef()` in React used for? | Refs / Advanced Patterns | Intermediate

`forwardRef()` is a React API that allows a component to forward its ref to one of its child components. This is useful when the parent component needs direct access to a DOM element or a child component's instance.

Explanation:

Normally, refs are used to directly access DOM elements or components. However, when you wrap a component, its ref doesn't automatically get passed down. `forwardRef()` lets you explicitly forward the ref to a specific child, enabling scenarios like focusing inputs or managing animations.

Example:


```
import React, { forwardRef, useRef } from 'react';

const CustomInput = forwardRef((props, ref) => {
  return <input ref={ref} {...props} />;
});

function App() {
  const inputRef = useRef();

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <CustomInput ref={inputRef} placeholder="Type here" />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}

export default App;
```

Here, `CustomInput` forwards the ref to the `<input>` element, enabling `App` to call `focus()`.

When to use:

- When you need to expose a DOM element or instance to a parent component.
- For building reusable input components that need focus or selection management.
- When integrating with third-party libraries that require direct DOM manipulation.

Mistakes to avoid:

- Forgetting to forward the ref and expecting it to work automatically.
- Using `forwardRef()` when it's unnecessary, adding complexity.
- Not handling the ref correctly in functional components.

Best practices:

- Use `forwardRef()` for wrapper components that need to expose internal elements.
- Document how the forwarded ref is used in the component's API.
- Combine `forwardRef()` with `useImperativeHandle` when you need to expose custom instance methods.

Interview scenarios:

- Asked to explain how to control a child input from the parent.
- Asked to implement reusable form fields with ref forwarding.
- Asked to expose methods in functional components.

---

22. How do you reset a component's state in React? | State Management | Beginner

Resetting a component's state means restoring it to its initial values, often used after form submissions or when navigating between views.

Explanation:

State in React is mutable through `setState` or the `useState` hook. To reset it, you simply assign the initial state again. This ensures the component's UI reflects the reset state.

Example:

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

const initialState = { name: '', email: '' };

function Form() {
  const [formData, setFormData] = useState(initialState);

  const handleReset = () => {
    setFormData(initialState);
  };

  return (
    <div>
      <input
        value={formData.name}
        onChange={(e) => setFormData({ ...formData, name: e.target.
        placeholder="Name"
      />
      <input
        value={formData.email}
        onChange={(e) => setFormData({ ...formData, email: e.target
        placeholder="Email"
      />
      <button onClick={handleReset}>Reset</button>
```

```
      </div>
    );
  }

  export default Form;
```

Clicking the reset button clears the form fields.

When to use:

- After submitting or canceling a form.
- When switching between different data views.
- For clearing controlled inputs.

Mistakes to avoid:

- Mutating the state directly instead of using `setState`.
- Using stale state when resetting due to incorrect reference copying.
- Forgetting to reset nested objects properly.

Best practices:

- Store initial state in a constant and reuse it.
- Avoid mutating state; always create a new object when resetting.
- Ensure form fields are controlled with `value` and `onChange`.

Interview scenarios:

- Asked to implement a form that can be reset.

- Asked how to manage controlled form inputs.
- Asked about state immutability and how to ensure state is updated correctly.

---

23. Why does React recommend against mutating state? | State Management / Best Practices | Intermediate

React's rendering system relies on detecting changes in state. If state is mutated directly, React may not detect changes, causing rendering bugs and stale UI.

Explanation:

React uses shallow comparison to determine if state or props have changed. When you mutate state directly, you don't create a new reference, so React may not rerender the component even though data has changed.

Example of incorrect approach:

```
const [items, setItems] = useState([1, 2, 3]);

function addItem() {
  items.push(4); // Direct mutation!
  setItems(items); // React might not rerender
}
```

Correct approach:

```
function addItem() {
  setItems(prevItems => [...prevItems, 4]); // Create new array
}
```

Why mutation causes issues:

- React compares old and new state by reference.
- Mutated state keeps the same reference, causing skipped renders.
- May lead to bugs where UI doesn't update correctly.

When this becomes a problem:

- Complex state objects or arrays.
- Nested state that's deeply updated.
- Optimized components using `React.memo`.

Mistakes to avoid:

- Mutating arrays with `push` or `splice` instead of creating new arrays.
- Mutating objects by assigning properties directly.
- Mixing mutable and immutable patterns.

Best practices:

- Always create a new object or array when updating state.
- Use helper functions like `map`, `filter`, or `spread syntax`.
- Embrace immutability for predictable rendering and debugging.

Interview scenarios:

- Asked why state changes sometimes don't reflect in the UI.
- Asked to identify why a component doesn't rerender after updating state.
- Asked how immutability helps in debugging or optimizing performance.

24. What are error boundaries in React for? | Error Handling | Intermediate

Error boundaries are special React components that catch errors in their child components during rendering, lifecycle methods, or constructors, preventing the entire app from crashing.

Explanation:

React's component tree doesn't handle errors by default. Error boundaries act as a safeguard by catching JavaScript errors and showing fallback UI instead of breaking the entire application.

How they work:

- Implemented using `componentDidCatch` and `getDerivedStateFromError`.
- Catch errors in rendering, lifecycle methods, and constructors.
- Don't catch errors in event handlers, asynchronous code, or server-side rendering.

Example:

```
import React from 'react';

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
```

```
    console.error("Error caught:", error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong.</h2>;
    }
    return this.props.children;
  }
}

function BuggyComponent() {
  throw new Error("Oops!");
  return <div>Normal content</div>;
}

function App() {
  return (
    <ErrorBoundary>
      <BuggyComponent />
    </ErrorBoundary>
  );
}

export default App;
```

When to use:

- Catch errors in UI rendering.
- Avoid breaking the entire app due to a minor component failure.
- Provide fallback UI to improve user experience.

Mistakes to avoid:

- Expecting error boundaries to catch errors in event handlers.
- Forgetting to log or report the error for debugging.

- Wrapping too large portions of the app, making error isolation harder.

Best practices:

- Wrap only critical parts of the UI.
- Use multiple error boundaries to localize errors.
- Provide meaningful fallback UI and logging.

Interview scenarios:

- Asked how to improve app resilience when parts of the UI fail.
- Asked to explain how React handles exceptions in child components.
- Asked about best practices in debugging production errors.

---

25. How do you test React applications? | Testing / Quality Assurance | Intermediate

Testing ensures that React components behave as expected. You can use libraries like Jest and React Testing Library to write unit, integration, and end-to-end tests.

Explanation:

Testing helps catch bugs early and ensures UI correctness. Jest is a test runner and assertion library, while React Testing Library simulates user interactions and verifies component behavior without relying on implementation details.

Types of tests:

1. **Unit tests:** Test individual components or functions.
2. **Integration tests:** Verify how components interact together.

3. **End-to-end tests:** Test full workflows, often using tools like Cypress.

Example (Unit test with React Testing Library):

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

test('increments count on button click', () => {
  render(<Counter />);
const button = screen.getByText('Increment');
fireEvent.click(button);
expect(screen.getByText('Count: 1')).toBeInTheDocument();
});
```

When to test:

- Before releasing features.
- When fixing bugs.
- While refactoring code.

Mistakes to avoid:

- Testing implementation details instead of behavior.
- Writing brittle tests that break easily with minor refactoring.
- Not cleaning up mocks or listeners between tests.

Best practices:

- Write tests that focus on what the user sees and interacts with.
- Use `beforeEach` and `afterEach` to reset state.
- Avoid over-mocking unless necessary.

Interview scenarios:

- Asked how to test a form submission flow.
- Asked to explain the difference between unit and integration tests.
- Asked why testing UI components improves maintainability.

---

26. Explain what React hydration is | Rendering / SSR | Intermediate

Hydration is the process where a server-rendered HTML page is “wired up” with JavaScript to make it interactive on the client side.

Explanation:

When you use Server-Side Rendering (SSR), the server sends a fully rendered HTML page to the browser. This improves load time and SEO. However, the HTML alone isn’t interactive event listeners and state management need to be attached once the JavaScript code is loaded on the client. This process of attaching event handlers and initializing state without replacing the existing markup is called hydration.

How it works:

1. Server renders the page and sends HTML.
2. The browser loads the HTML and displays it.
3. React runs on the client, finds the matching DOM structure, and attaches event listeners without re-rendering.

Example with Next.js:

```
// pages/index.js
function Home() {
  const [count, setCount] = useState(0);

  return (
```

```
      <div>
        <h1>Welcome</h1>
        <button onClick={() => setCount(count + 1)}>
          Count: {count}
        </button>
      </div>
    );
  }

  export default Home;
```

This page can be server-rendered by Next.js and hydrated on the client to become interactive.

When hydration is important:

- SEO-focused websites that benefit from server rendering.
- Apps where fast initial load is critical.
- Complex UIs that need both pre-rendered content and interactivity.

Mistakes to avoid:

- Hydration mismatch errors when server and client render differently.
- Adding dynamic data on the server that changes on the client without syncing.
- Not using consistent props or markup between server and client.

Best practices:

- Ensure that server-rendered markup matches client-side rendering.
- Avoid using non-deterministic data during initial render.
- Handle loading and fallback states carefully during hydration.

Interview scenarios:

- Asked about the difference between SSR and CSR.
- Asked how hydration improves performance.
- Asked why hydration errors occur and how to debug them.

---

27. What are React Portals used for? | Rendering / Advanced Patterns | Intermediate

Portals allow you to render a component's children into a DOM node that exists outside the parent component's hierarchy.

Explanation:

Sometimes you need UI elements like modals, tooltips, or dropdowns to break out of their parent container's styles, overflow, or z-index constraints. Portals let you render content elsewhere in the DOM tree while keeping the component's logic within its parent.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom';

function Modal({ children }) {
  return ReactDOM.createPortal(
    <div className="modal">
      {children}
    </div>,
    document.getElementById('modal-root') // outside the main app c
  );
}

function App() {
```

```
    return (
      <div>
        <h1>My App</h1>
        <Modal>
          <h2>I'm a modal!</h2>
        </Modal>
      </div>
    );
  }

  export default App;
```


Here, `Modal` is rendered into a separate `div` with ID `modal-root`, outside the main app tree.

When to use:

- Building modals, popovers, or tooltips.
- Managing z-index or overflow issues.
- Isolating UI elements that require special handling.

Mistakes to avoid:

- Forgetting to create the target DOM node (`modal-root`).
- Rendering mismatched content between parent and portal.
- Not handling accessibility concerns (like focus trapping).

Best practices:

- Ensure the portal's target node exists before rendering.
- Manage focus and accessibility within portals.

- Avoid passing complex state unless needed.

Interview scenarios:

- Asked how to build a modal component.
- Asked how to solve overflow or styling issues in nested layouts.
- Asked how to manage event bubbling with portals.

---

28. How do you debug React applications? | Debugging / Tools | Beginner

Debugging in React involves inspecting component trees, checking state and props, and identifying errors using browser tools and libraries.

Explanation:

Debugging helps developers find and fix issues during development or in production. React apps can be debugged using both built-in browser tools and external libraries designed to inspect React-specific structures.

Tools and methods:

1. **React Developer Tools:** Inspect the component hierarchy, state, and props.
2. **Console logs:** Use `console.log()` to trace state updates or errors.
3. **Breakpoints:** Set breakpoints in browser DevTools to step through code.
4. **Error boundaries:** Catch rendering errors and inspect stack traces.
5. **Network inspection:** Check API requests and responses.

Example use of React Developer Tools:

- Inspect a component's props.
- See which component caused an error.
- Trace state changes and understand component rerenders.

Common debugging approaches:

```
console.log("Current state:", state);
console.error("Something went wrong!");
```


Use conditionals to debug state transitions or logic flow.

Mistakes to avoid:

- Logging too much data, making debugging harder.
- Relying only on `console.log` instead of structured error handling.
- Forgetting to clean up event listeners or intervals during component unmount.

Best practices:

- Use React DevTools for structural inspection.
- Leverage `useEffect` cleanup functions to manage side effects.
- Integrate monitoring tools like Sentry in production environments.

Interview scenarios:

- Asked how you would identify why a component isn't updating.
- Asked how you would debug API calls in a React app.
- Asked how you would troubleshoot hydration issues.

29. What is React strict mode and what are its benefits? | Development Tools / Best Practices | Beginner

React Strict Mode is a tool that helps developers find potential issues by adding additional checks and warnings during development.

Explanation:

Strict Mode doesn't render anything on the screen, but it activates extra checks in the background. It helps catch unsafe lifecycle methods, side effects, deprecated APIs, and other common issues early.

Key features:

- Detects components with unsafe lifecycle methods.
- Identifies side effects and unexpected behaviors.
- Highlights deprecated APIs.
- Helps prepare code for future versions of React.

Example usage:

```
import React from 'react';

function App() {
  return (
    <React.StrictMode>
      <MyComponent />
    </React.StrictMode>
  );
}
```

This wraps the component and applies extra checks during development.

When to use:

- During development to catch bugs early.
- To prepare for future React updates.
- To ensure best practices in component design.

Mistakes to avoid:

- Thinking it's meant for production performance optimization.
- Ignoring warnings without fixing underlying issues.
- Confusing it with error boundaries (they serve different purposes).

Best practices:

- Always enable it during development.
- Use it alongside testing and code reviews.
- Address warnings to maintain code health.

Interview scenarios:

- Asked how to ensure code quality in a React app.
- Asked how to find side effects or deprecated code.
- Asked why a component behaves unexpectedly in strict mode.

---

30. How do you localize React applications? | Internationalization (i18n) | Intermediate

Localization is the process of adapting an application's content for different languages and regions, making it accessible to a broader audience.

Explanation:

Localization involves providing translations, date/time formats, number formats, and culturally appropriate content. In React, libraries like `react-i18next` and `react-intl` make it easier to manage translations and switch languages dynamically.

Steps to localize an app:

1. Create translation files for different languages.
2. Use an i18n library to load and switch translations.
3. Replace static text with translatable keys.

Example using `react-i18next`:

```
// i18n.js
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';

i18n.use(initReactI18next).init({
  resources: {
    en: {
      translation: {
        welcome_message: "Welcome!",
      },
    },
    fr: {
      translation: {
        welcome_message: "Bienvenue!",
      },
    },
  },
  lng: "en",
```

```
  fallbackLng: "en",
});

// Component
import { useTranslation } from 'react-i18next';

function MyComponent() {
  const { t } = useTranslation();
  return <p>{t('welcome_message')}</p>;
}
```

When to use:

- Applications targeting global audiences.
- Apps with dynamic language switching.
- Apps needing support for multiple date, time, or currency formats.

Mistakes to avoid:

- Hardcoding text instead of using translation keys.
- Forgetting to handle fallback languages.
- Not considering text direction or pluralization rules.

Best practices:

- Store translations in separate files for maintainability.
- Use context-aware translations for complex sentences.
- Test with multiple languages to ensure correct rendering.

Interview scenarios:

- Asked how you would support multiple languages in a web app.

- Asked how to manage translations for dynamic content.
- Asked how to ensure fallback options when translations are missing.

---

31. What is code splitting in a React application? | Performance Optimization | Intermediate

Code splitting is a technique that helps improve performance by breaking the application's code into smaller chunks and loading them on demand.

Explanation:

In large React applications, bundling everything into one file can slow down the initial load time. Code splitting allows you to split your code at logical points (such as routes or components) so that only the necessary code is loaded when required.

React provides built-in support for code splitting through `React.lazy()` and `Suspense`, allowing developers to load components asynchronously.

How it works:

1. Define a component to be lazily loaded using `React.lazy`.
2. Wrap the component with `React.Suspense`, specifying a fallback UI while the component loads.

Example:

```
import React, { Suspense, lazy } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <h1>Main App</h1>
```

```
        <Suspense fallback={<div>Loading...</div>}>
          <LazyComponent />
        </Suspense>
      </div>
    );
  }

  export default App;
```

Here, `LazyComponent` is loaded only when needed, improving the initial load time.

When to use:

- Large applications with multiple routes.
- Components not required at the initial render.
- Reducing the size of JavaScript bundles.

Mistakes to avoid:

- Not wrapping lazy components with `Suspense` , leading to errors.
- Overusing code splitting, causing excessive network requests.
- Not handling errors during dynamic imports.

Best practices:

- Code split by route or feature rather than at arbitrary points.
- Use fallback UI that provides useful loading feedback.
- Combine with error boundaries to handle loading failures gracefully.

Interview scenarios:

- Asked how to optimize performance for large React apps.
- Asked why code splitting is beneficial.
- Asked how to handle lazy loading errors in production.

---

32. How would one optimize the performance of React contexts to reduce rerenders? | State Management / Performance | Advanced

Optimizing React context involves ensuring that only the necessary components re-render when context values change.

Explanation:

React's context API provides a way to pass data through the component tree without prop drilling. However, when context values change, all components that consume the context re-render by default, even if they don't use the changed part of the context.

Optimizing context ensures better performance by minimizing unnecessary re-renders.

Techniques to optimize:

1. **Memoize context values:**
   Use `useMemo` to prevent context values from changing unnecessarily.

```
const value = useMemo(() => ({ state, dispatch }), [state, dispatch
```

2. **Split contexts:**
   Create separate contexts for unrelated state slices.

3. **Use selectors:**
   Libraries like `use-context-selector` allow consumers to only listen to specific parts of the context.

Example:

```
const AppContext = React.createContext();

function AppProvider({ children }) {
  const [count, setCount] = useState(0);
  const value = useMemo(() => ({ count, setCount }), [count]);

  return (
    <AppContext.Provider value={value}>
      {children}
    </AppContext.Provider>
  );
}
```


When to apply:

- Applications with deep component trees.
- State shared across unrelated components.
- Performance-critical apps where frequent state updates occur.

Mistakes to avoid:

- Passing inline objects or functions as context values, triggering re-renders.
- Using a single context for all state, causing widespread updates.
- Forgetting to memoize or separate unrelated state.

Best practices:

- Structure contexts based on state usage patterns.
- Always memoize values that change over time.
- Avoid unnecessary context usage when local state suffices.

Interview scenarios:

- Asked why your app's performance drops with global state.
- Asked how to manage global state in a scalable way.
- Asked about alternatives to Redux or MobX.

---

33. What are higher order components in React? | Component Patterns | Intermediate

Higher-order components (HOCs) are functions that take a component as input and return a new component with added functionality or props.

Explanation:

HOCs are a design pattern for reusing logic across components without duplicating code. Instead of inheritance, you wrap a component to enhance its behavior.

A typical HOC adds props, handles state, or injects logic into wrapped components.

Example:

```
const withLogger = (WrappedComponent) => {
  return (props) => {
    console.log('Rendering with props:', props);
    return <WrappedComponent {...props} />;
  };
};
```

```
const Hello = ({ name }) => <div>Hello, {name}!</div>;

const HelloWithLogger = withLogger(Hello);
```

Now, `HelloWithLogger` logs props before rendering `Hello`.

When to use:

- Reusing component logic like authentication checks or data fetching.
- Injecting props without modifying the original component.
- Enhancing components for testing or logging.

Mistakes to avoid:

- Mutating props or state inside the HOC.
- Not forwarding refs when needed.
- Using HOCs where hooks would suffice, complicating the component tree.

Best practices:

- Use HOCs for shared logic that applies to multiple components.
- Avoid overusing them; prefer hooks where appropriate.
- Ensure composability by properly handling props and refs.

Interview scenarios:

- Asked to explain patterns for code reuse.
- Asked how you would share logic across multiple components.

- Asked why inheritance is avoided in React.

---

34. What is the Flux pattern and what are its benefits? | State Management | Intermediate

Flux is an architectural pattern for managing application state with a unidirectional data flow.

Explanation:

In traditional applications, data can flow in multiple directions, making it hard to track state changes. Flux enforces a single direction for data flow, making state management predictable and easier to debug.

Core components:

1. **Dispatcher:** Sends actions to stores.
2. **Stores:** Hold application state and logic.
3. **Actions:** Define events that change the state.
4. **Views (Components):** React to changes in stores and update the UI.

Data flow sequence:

1. User triggers an event.
2. An action is created and dispatched.
3. Stores update based on the action.
4. Views re-render with updated state.

Benefits:

- Predictable state transitions.
- Easier debugging and tracing of state changes.
- Separation of concerns between data and UI logic.

Example illustration:

```
// Action
const incrementAction = { type: 'INCREMENT' };

// Store
let count = 0;
function handleAction(action) {
  switch(action.type) {
    case 'INCREMENT':
      count += 1;
      break;
  }
}
```

This simplified model shows how actions modify state and trigger UI updates.

Mistakes to avoid:

- Overcomplicating the dispatcher or stores.
- Using it for trivial state instead of shared state.
- Mixing responsibilities across actions and stores.

Best practices:

- Keep actions descriptive and minimal.
- Ensure stores only hold state logic.

- Use libraries like Redux that implement Flux principles with added tools.

Interview scenarios:

- Asked why unidirectional data flow helps in complex apps.
- Asked how Redux or MobX relate to Flux.
- Asked to describe how state changes propagate in your app.

---

35. Explain one-way data flow of React and its benefits | Data Flow / Core Concepts | Beginner

One-way data flow means that data flows from parent components down to child components, making state management more predictable.

Explanation:

In React, data flows in a single direction—from parent components to child components via props. This ensures that data dependencies are clear and that child components cannot directly modify data owned by parent components.

How it works:

- Parent defines state and passes it to children.
- Children render UI based on props.
- Children communicate changes via callbacks passed from the parent.

Example:

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

```
  return (
    <Child count={count} increment={() => setCount(count + 1)} />
  );
}

function Child({ count, increment }) {
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```


Here, `Parent` manages state, and `Child` uses props to render and request updates.

Benefits:

- Predictable UI updates.
- Easier debugging since data flow is traceable.
- Reduces unexpected side effects.
- Encourages better component design and separation of concerns.

Mistakes to avoid:

- Passing unnecessary props that cause re-renders.
- Overusing prop drilling instead of context for global state.
- Allowing children to mutate parent data directly.

Best practices:

- Keep state where it logically belongs.

- Use callbacks to notify parents of changes.
- Use context or state managers when prop drilling gets cumbersome.

Interview scenarios:

- Asked how state updates in React are managed.
- Asked why data shouldn't be passed in multiple directions.
- Asked how to handle shared state across components.

---

36. How do you handle asynchronous data loading in React applications? | Data Fetching / Side Effects | Intermediate

Handling asynchronous data loading in React involves fetching data from APIs or other sources and updating the component state accordingly while managing loading and error states.

Explanation:

In React, data fetching is typically done inside the `useEffect` hook, which allows you to run side effects after the component mounts or when dependencies change. You also use `useState` to store the data and track the loading or error status.

Steps to handle data fetching:

1. Define state variables to store data, loading, and error.
2. Use `useEffect` to trigger the data fetch on component mount or when dependencies change.
3. Use `async/await` or promises to fetch data.
4. Update state when data is successfully loaded or an error occurs.

Example:

```
import React, { useState, useEffect } from 'react';

function DataFetchingComponent() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch('https://api.example.com/data'
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    }

    fetchData();
  }, []); // Empty dependency array to run once on mount

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return <div>Data: {JSON.stringify(data)}</div>;
}
```

When to apply:

- Loading data from REST APIs.

- Interacting with backend services.
- Updating UI based on external data sources.

Mistakes to avoid:

- Forgetting the empty dependency array, causing infinite fetch loops.
- Not handling error states, leading to broken UI.
- Updating state after the component has unmounted.

Best practices:

- Always handle errors gracefully.
- Cancel or clean up requests when the component unmounts.
- Use loading states to inform users while data is being fetched.

Interview scenarios:

- Asked how you would fetch data in a functional component.
- Asked how to avoid memory leaks in components.
- Asked how to manage loading and error states.

---

37. Explain server-side rendering of React applications and its benefits | Rendering / Performance | Intermediate

Server-side rendering (SSR) refers to rendering React components on the server and sending fully rendered HTML to the client.

Explanation:

Normally, React applications are client-rendered, where JavaScript loads and renders the UI after the page has loaded. With SSR, the server pre-renders the page into HTML, improving performance and SEO.

After the initial HTML is sent to the browser, React takes over with hydration, attaching event listeners and enabling interactivity.

Benefits:

1. **Improved SEO:** Search engines can index content from the initial HTML.
2. **Faster load times:** Users see content faster since it's rendered on the server.
3. **Better performance on slow devices:** Less JavaScript processing is required on the client.

How SSR works:

1. Server renders components into static HTML.
2. HTML is sent to the client.
3. React hydrates the page, making it interactive.

Example using Next.js:

```
// pages/index.js
export async function getServerSideProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();
  return { props: { data } };
}

export default function Home({ data }) {
  return <div>Data: {JSON.stringify(data)}</div>;
}
```

Here, `getServerSideProps` fetches data during each request on the server.

Mistakes to avoid:

- Fetching data incorrectly without server context.
- Rendering components that rely on browser-specific APIs without checks.
- Not using caching strategies, causing slow responses.

Best practices:

- Handle browser-only code using feature detection or lifecycle checks.
- Cache frequently requested data where possible.
- Use frameworks like Next.js that provide built-in SSR support.

Interview scenarios:

- Asked why SEO is a problem with client-side rendering.
- Asked how you would implement SSR in your app.
- Asked to compare SSR vs static generation.

---

38. Explain static generation of React applications and its benefits | Rendering / Performance | Intermediate

Static generation refers to pre-rendering pages at build time rather than on each request, delivering optimized HTML files.

Explanation:

Static generation builds the pages ahead of time into HTML files that can be served quickly, often from a CDN. It’s suitable for pages that don’t change frequently and

helps achieve fast load times and better SEO.

Frameworks like Next.js allow developers to generate static pages easily with functions like `getStaticProps`.

Benefits:

1. **Blazing fast load times:** Pages are served without runtime data fetching.
2. **SEO-friendly content:** Search engines see the full content without JavaScript execution.
3. **Lower server load:** Pages are generated at build time.

How it works:

1. At build time, data is fetched and pages are pre-rendered.
2. The static files are deployed to servers or CDNs.
3. Clients receive ready-to-render HTML.

Example with Next.js:

```
// pages/index.js
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();
  return { props: { data } };
}

export default function Home({ data }) {
  return <div>Data: {JSON.stringify(data)}</div>;
}
```

The data is fetched during the build process.

When to use:

- Marketing pages, blogs, documentation sites.
- Pages with infrequent content updates.
- Sites aiming for performance and SEO improvements.

Mistakes to avoid:

- Using static generation for highly dynamic pages.
- Not implementing incremental regeneration when needed.
- Failing to handle API errors at build time.

Best practices:

- Use static generation where data rarely changes.
- Combine with incremental static regeneration for freshness.
- Optimize API calls to reduce build time.

Interview scenarios:

- Asked how to optimize page load times.
- Asked the difference between SSR and static generation.
- Asked when to prefer static generation over SSR.

---

39. Explain the presentational vs container component pattern in React | Component Architecture | Intermediate

This pattern helps separate concerns by distinguishing between components responsible for UI and those responsible for logic and state.

Explanation:

- **Presentational components:** Focus on how the UI looks. They receive data and callbacks via props and have little to no state.
- **Container components:** Focus on how the application works. They manage state, data fetching, and logic, and pass props to presentational components.

This separation makes code more reusable, testable, and maintainable.

Example:

```
// Presentational component
function UserList({ users }) {
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

// Container component
function UserContainer() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch('/api/users')
      .then((res) => res.json())
      .then((data) => setUsers(data));
  }, []);
```

```
    return <UserList users={users} />;
  }
```

Here, `UserList` only renders the UI, while `UserContainer` handles fetching data.

Benefits:

1. Clear separation of UI and logic.
2. Reusable and easily testable components.
3. Encourages a modular and scalable architecture.

Mistakes to avoid:

- Mixing state logic into presentational components.
- Overusing containers, making code harder to navigate.
- Ignoring prop drilling when it’s unnecessary.

Best practices:

- Keep presentational components stateless where possible.
- Move logic to containers for better readability.
- Use hooks to manage side effects in containers.

Interview scenarios:

- Asked how you structure components in a large project.
- Asked how you make components reusable.
- Asked to explain separation of concerns in React.

40. What are some common pitfalls when doing data fetching in React? | Data Fetching / Error Handling | Intermediate

Data fetching can lead to various issues if not handled correctly, impacting performance, usability, or causing bugs.

Explanation:

While fetching data in React is straightforward, developers often overlook edge cases like race conditions, memory leaks, or improper error handling. Recognizing and avoiding these pitfalls ensures robust and performant applications.

Common pitfalls:

1. **Not handling loading and error states:**
   Users might see incomplete or broken interfaces.

2. **Ignoring cleanup of subscriptions:**
   Leads to memory leaks when components unmount before the fetch completes.

3. **Infinite loops in `useEffect`:**
   Caused by missing dependency arrays or including unstable references.

4. **Directly mutating state:**
   React won't track changes if state is updated improperly.

5. **Using stale closures:**
   Functions capturing outdated state can lead to unexpected results.

Best practices:

- Use proper dependency arrays in `useEffect`.
- Use `AbortController` to cancel fetch requests on unmount.
- Handle loading and error states explicitly.

- Avoid updating state after a component unmounts.
- Structure data fetching logic using custom hooks.

Example with cleanup:

```
import React, { useState, useEffect } from 'react';

function DataComponent() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const controller = new AbortController();

    fetch('/api/data', { signal: controller.signal })
      .then((res) => res.json())
      .then((data) => {
        setData(data);
        setLoading(false);
      })
      .catch((err) => {
        if (err.name !== 'AbortError') {
          console.error('Fetch error:', err);
        }
      });

    return () => controller.abort(); // Cleanup on unmount
  }, []);

  if (loading) return <div>Loading...</div>;
  return <div>{JSON.stringify(data)}</div>;
}
```

Interview scenarios:

- Asked how to prevent memory leaks.

- Asked how to handle component unmounting during data fetches.
- Asked to explain error handling strategies.

---

41. What is the role of keys in React lists and why are they important? | Rendering / Lists | Beginner

Keys help React identify which items have changed, been added, or removed when rendering lists.

Explanation:

When rendering lists in React, each element needs a unique `key` so that React can efficiently update only the parts that have changed. Without keys, React will re-render the entire list unnecessarily, causing performance issues and potentially leading to UI bugs.

How it works:

- Keys are used by React's reconciliation algorithm to match elements between renders.
- If keys change or are missing, React may reuse incorrect elements, leading to unexpected behavior.

Example:

```
const items = ['Apple', 'Banana', 'Cherry'];

function ItemList() {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={item}>{item}</li>
      ))}
```

```
      </ul>
    );
  }
```

Here, `item` is used as the key because it's unique for each list element.

When to use:

- Rendering dynamic lists like menus, tables, or grids.
- Handling updates where items are added or removed.

Mistakes to avoid

- Using array index as key in dynamic lists — this can cause incorrect updates.
- Not using unique keys, leading to bugs in animations or state management.
- Changing keys unnecessarily, causing components to unmount and remount.

Best practices:

- Use stable, unique IDs as keys.
- Avoid using array indices if the list can change.
- Ensure keys remain consistent between renders.

Interview scenarios:

- Asked how React optimizes list rendering.
- Asked why using index as a key can cause issues.
- Asked how to handle updates in large lists.

---

42. What are fragments in React and why are they useful? | Rendering / JSX | Beginner

Fragments let you group multiple elements without adding extra nodes to the DOM.

Explanation:

In React, a component must return a single element. Sometimes you need to group elements without introducing unnecessary markup, like extra `<div>` s that clutter the HTML structure. React fragments solve this by allowing you to wrap elements without adding new DOM nodes.

Example:

```
import React from 'react';

function Table() {
  return (
    <>
      <tr>
        <td>Row 1, Cell 1</td>
        <td>Row 1, Cell 2</td>
      </tr>
      <tr>
        <td>Row 2, Cell 1</td>
        <td>Row 2, Cell 2</td>
      </tr>
    </>
  );
}
```

This example groups multiple rows without adding an extra wrapper.

When to use:

- Returning multiple elements from a component.

- Avoiding unnecessary DOM nodes.
- Grouping elements inside lists or table rows.

Mistakes to avoid:

- Adding unnecessary wrapper elements and complicating styling.
- Forgetting that fragments don't create actual DOM elements, which might affect selectors or CSS.

Best practices:

- Use `<>...</>` syntax for simpler code.
- Use `<React.Fragment>` when keys are needed for lists.
- Avoid over-wrapping content unnecessarily.

Interview scenarios:

- Asked how to return multiple elements from a component.
- Asked how to avoid unnecessary markup in the DOM.
- Asked about performance implications of extra DOM nodes.

---

43. What are controlled and uncontrolled components in React? | Forms / State | Intermediate

Controlled components have their state managed by React, while uncontrolled components manage their own state internally.

Explanation:

- **Controlled components:** The value of the input is controlled via React's state. The component's data is always in sync with the state.

- **Uncontrolled components:** The input's value is handled by the DOM itself, and React accesses it using refs when needed.

Example of a controlled component:

```
function ControlledInput() {
  const [value, setValue] = React.useState('');

  return (
    <input
      type="text"
      value={value}
      onChange={(e) => setValue(e.target.value)}
    />
  );
}
```

Example of an uncontrolled component:

```
function UncontrolledInput() {
  const inputRef = React.useRef();

  const handleSubmit = () => {
    alert('Input value: ' + inputRef.current.value);
  };

  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
}
```

When to use:

- Controlled: When you need to validate input, manage forms, or synchronize state.
- Uncontrolled: When simple forms or quick prototypes don't need state syncing.

Mistakes to avoid:

- Mixing controlled and uncontrolled logic in the same form.
- Forgetting to update state in controlled components, causing stale inputs.
- Overusing uncontrolled components in complex forms.

Best practices:

- Use controlled components for forms with validation or interactions.
- Use uncontrolled components for simpler cases.
- Keep form logic centralized and testable.

Interview scenarios:

- Asked how to manage form inputs in React.
- Asked when to use refs vs state.
- Asked to explain the differences and trade-offs between the two.

---

44. Explain the use of the Context API in React | State Management | Intermediate

The Context API provides a way to share state across the component tree without passing props through every level.

Explanation:

React's Context API allows data to be passed globally through the component hierarchy, making it useful for themes, user data, or settings that multiple components need to access.

It helps avoid "prop drilling," where props are passed through many intermediate components unnecessarily.

Steps to use Context:

1. Create a context.
2. Wrap components with a provider.
3. Use the context in child components via `useContext`.

Example:

```
import React, { createContext, useContext, useState } from 'react';

const ThemeContext = createContext();

function App() {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return <ThemedButton />;
}

function ThemedButton() {
```

```
  const { theme } = useContext(ThemeContext);
  return <button>{`Current theme: ${theme}`}</button>;
}
```

When to use:

- Sharing data like themes, authentication, or settings.
- Avoiding excessive prop passing.
- Managing global state in smaller applications.

Mistakes to avoid:

- Using context for frequently changing data, causing unnecessary rerenders.
- Overusing context when local state would suffice.
- Not memoizing context values.

Best practices:

- Separate concerns into multiple contexts if needed.
- Memoize context values with `useMemo`.
- Avoid heavy computations inside context providers.

Interview scenarios:

- Asked how to avoid prop drilling.
- Asked how you would share data across unrelated components.
- Asked to explain when to use context vs Redux.

---

45. What are React hooks and why were they introduced? | Core Concepts | Beginner

Hooks are functions that let you use React features like state and lifecycle methods in functional components.

Explanation:

Before hooks, state and lifecycle logic could only be used in class components. Hooks were introduced to enable functional components to manage state, side effects, and other behaviors without the complexity of classes.

This simplified code, improved reuse, and aligned better with JavaScript functions.

Popular hooks:

- `useState` : Manage state in functional components.
- `useEffect` : Handle side effects like data fetching or subscriptions.
- `useContext` : Access global data.
- `useRef` : Persist mutable values or DOM references.
- `useReducer` : Manage complex state logic.

Example using `useState` :

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

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

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

```
      </div>
    );
  }
```

Why hooks were introduced:

1. Avoid class-based complexity.
2. Reuse logic across components via custom hooks.
3. Simplify stateful logic in functional components.
4. Improve readability and testability.

Mistakes to avoid:

- Calling hooks conditionally or inside loops.
- Forgetting to follow the rules of hooks.
- Not understanding dependency arrays in `useEffect`.

Best practices:

- Always call hooks at the top level of the component.
- Use custom hooks to extract shared logic.
- Use dependency arrays carefully and intentionally.

Interview scenarios:

- Asked why hooks are preferred over class components.
- Asked how you would refactor class logic into hooks.

- Asked to explain rules and restrictions when using hooks.

---

46. How does the virtual DOM in React work? What are its benefits and downsides? | Rendering / Performance | Intermediate

The virtual DOM is a lightweight copy of the actual DOM that helps React efficiently update the UI by comparing changes before applying them to the real DOM.

Explanation:

When the state or props of a component change, React doesn't update the actual DOM immediately. Instead, it creates a virtual representation of the DOM, compares it with the previous version, and identifies what's different. This process, known as "diffing" or "reconciliation," allows React to update only the necessary parts, improving performance.

Steps in virtual DOM:

1. A state or prop change triggers a re-render.
2. A new virtual DOM tree is created based on the updated component.
3. The new virtual DOM is compared with the previous version.
4. Differences are calculated.
5. Only the changed parts are updated in the actual DOM.

Benefits:

- **Improved performance:** Minimizes direct DOM manipulations which are expensive.
- **Declarative UI:** Developers only describe how the UI should look.

- **Predictability:** Updates are applied systematically, reducing bugs.

Downsides:

- **Overhead:** Creating and comparing virtual DOM trees adds computational work.
- **Not always faster:** For very simple or static applications, virtual DOM may not offer significant benefits.
- **Requires understanding:** Misusing state updates or improper patterns can negate performance advantages.

Example:

```
function Counter() {
  const [count, setCount] = React.useState(0);

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

Here, React compares the old and new virtual DOM before updating the count.

Mistakes to avoid:

- Assuming virtual DOM optimizations always make the app faster.
- Updating state too frequently without batching.
- Not structuring components efficiently, causing unnecessary renders.

Best practices:

- Break the UI into small, reusable components.
- Avoid unnecessary state changes.
- Use tools like React DevTools to inspect renders.

Interview scenarios:

- Asked how React optimizes rendering.
- Asked why manipulating the DOM directly is discouraged.
- Asked about the difference between virtual DOM and real DOM updates.

---

47. What is React Fiber and how is it an improvement over the previous approach? | Architecture / Performance | Advanced

React Fiber is a reimplementation of React's core algorithm that improves rendering performance and flexibility by allowing incremental rendering and better prioritization.

Explanation:

Before React 16, React used a stack-based algorithm that couldn't pause or resume rendering tasks. React Fiber introduced a new approach that breaks rendering into small units of work, enabling React to handle animations, gestures, and large trees more efficiently.

Key improvements:

- **Incremental rendering:** Large updates are split into smaller tasks that can be paused and resumed.
- **Prioritization:** Updates like animations can be given higher priority, while less important tasks are deferred.

- **Error boundaries and suspense:** Fiber makes it possible to handle errors and load states gracefully.
- **Time slicing:** React can now schedule work during idle time, improving responsiveness.

Example use cases:

- Complex animations with smooth transitions.
- Handling asynchronous operations in large applications.
- Graceful error recovery without crashing the entire app.

Mistakes to avoid:

- Assuming Fiber changes how you write React code—it's under the hood.
- Ignoring performance implications when managing state updates.
- Not optimizing component structure to take advantage of Fiber's scheduling.

Best practices:

- Use `React.Suspense` and `lazy` to manage code splitting.
- Avoid blocking the main thread with heavy computations.
- Structure components to minimize unnecessary re-rendering.

Interview scenarios:

- Asked why React 16 was a major upgrade.
- Asked about performance optimizations and rendering interruptions.
- Asked to explain how React handles animations and UI responsiveness.

48. What is reconciliation in React? | Rendering / Performance | Intermediate

Reconciliation is the process by which React compares two versions of the virtual DOM and updates the actual DOM efficiently.

Explanation:

When state or props change, React generates a new virtual DOM. It then compares the new tree with the previous one and determines the minimal set of changes required to update the actual DOM. This comparison process is called reconciliation.

How it works:

1. React receives new state or props.
2. A new virtual DOM tree is created.
3. The new tree is compared with the old one.
4. Differences are identified.
5. Only the changed elements are updated in the real DOM.

Example scenario:

```
const [count, setCount] = useState(0);
<button onClick={() => setCount(count + 1)}>Click me</button>
```

When `setCount` is triggered, React compares the previous count value and updates the text without re-rendering the entire tree.

Benefits:

- Avoids unnecessary DOM updates.
- Improves app performance.
- Makes UI updates predictable.

Mistakes to avoid:

- Assuming reconciliation always prevents performance issues.
- Using improper keys that confuse the algorithm.
- Not understanding how deep component trees affect performance.

Best practices:

- Use keys correctly to guide reconciliation.
- Structure components to localize state changes.
- Avoid complex data structures that complicate diffing.

Interview scenarios:

- Asked how React decides which elements to update.
- Asked why keys are important during updates.
- Asked about performance implications in dynamic lists.

---

49. What is React Suspense and what does it enable? | Async Handling / Performance | Advanced

React Suspense is a feature that allows you to handle asynchronous operations like data fetching or code splitting by showing fallback content while waiting for a resource.

Explanation:

Suspense helps manage asynchronous dependencies declaratively by letting React "wait" until the data or code is ready before rendering the component. This makes it easier to coordinate loading states across multiple components.

Core concepts:

- Wrap components with `React.Suspense`.
- Provide a fallback UI while waiting.
- Combine with `React.lazy` or data-fetching libraries.

Example:

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

function App() {
  return (
    <React.Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </React.Suspense>
  );
}
```

Here, `LazyComponent` is loaded asynchronously, and "Loading..." is displayed until it's ready.

Use cases:

- Code splitting to load components on demand.
- Handling slow data fetching in UI without showing blank states.
- Coordinating multiple async operations in a unified way.

Mistakes to avoid:

- Not wrapping lazy-loaded components inside `Suspense`.
- Providing overly complex fallback UIs that confuse users.
- Mixing Suspense with non-compatible libraries.

Best practices:

- Use simple and user-friendly fallback UIs.
- Combine with error boundaries to gracefully handle load failures.
- Apply Suspense only to parts of the UI that need it.

Interview scenarios:

- Asked how you would improve user experience during data loading.
- Asked to explain code splitting and how Suspense helps.
- Asked how to handle fallback states without disrupting the main UI.

---

50. Explain what happens when the `useState` setter function is called in React | State Management / Hooks | Intermediate

The `useState` setter schedules a state update, queues a re-render, and applies changes asynchronously.

Explanation:

When you call the setter function returned by `useState`, React schedules the state change and triggers a re-render of the component with the updated state. React batches multiple state updates for better performance and applies them before the next paint.

Detailed process:

1. The setter is called with a new state or function.
2. React queues the update.
3. During the next render phase, React updates the component's state.
4. React reconciles the virtual DOM and applies changes to the real DOM.
5. The UI reflects the new state.

Example:

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

  const increment = () => setCount(prev => prev + 1);

  return (
    <div>
      <p>{`Count: ${count}`}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

Here, `setCount` schedules the state change based on the previous state.

Why the functional form is important:

```
setCount(prevCount => prevCount + 1);
```

This ensures that you always update the state based on the latest value, especially when multiple updates are batched together.

Mistakes to avoid:

- Updating state with a stale value by directly referencing it.
- Calling the setter in loops without considering batching.
- Misunderstanding that state updates are asynchronous.

Best practices:

- Use the functional update form when new state depends on the previous state.
- Avoid unnecessary state updates that cause performance bottlenecks.
- Leverage batching for efficiency.

Interview scenarios:

- Asked what happens internally when state is updated.
- Asked why state updates may not reflect immediately.
- Asked how to manage state changes depending on previous values.

---

51. What is the difference between `React.memo` and `useMemo`? When would you use each? | Performance Optimization / Hooks | Advanced

- `React.memo` is a **higher-order component** that memoizes a **React component**, preventing it from re-rendering unless its props change.
- `useMemo` is a **hook** that memoizes a **computed value** inside a component, so the value is only recalculated when its dependencies change.

Example:

```
import React, { useMemo } from 'react';

const ExpensiveComponent = React.memo(({ data }) => {
  console.log('Rendering ExpensiveComponent');
  return <div>{data.value}</div>;
});

function App({ items }) {
  const computedValue = useMemo(() => {
    return items.reduce((acc, item) => acc + item.value, 0);
  }, [items]);

  return (
    <div>
      <ExpensiveComponent data={{ value: computedValue }} />
    </div>
  );
}
```

Key points:

- Use `React.memo` to optimize **component rendering**.
- Use `useMemo` to optimize **expensive calculations** inside a component.

Common pitfalls:

- Overusing memoization can **increase memory usage** without noticeable performance gain.
- `React.memo` only does shallow comparison by default. For deep props, use a custom comparison function.

Sample interview scenario:

- A candidate is asked to optimize a dashboard component that re-renders unnecessarily when unrelated state updates occur.

52. How does React handle events differently from native DOM events? | Event Handling | Intermediate

- React uses a **synthetic event system** to normalize events across browsers.
- Synthetic events are **pooled**, meaning React reuses them for performance, so the event object may be nullified after the callback finishes.
- Event delegation: React attaches a single listener at the root and delegates events to components, improving performance.

Example:

```
function Button() {
  const handleClick = (e) => {
    console.log(e.type); // "click"
    setTimeout(() => console.log(e.type), 1000); // e may be null d
  };
  return <button onClick={handleClick}>Click Me</button>;
}
```

Best practices:

- If you need to use the event asynchronously, call `e.persist()` to retain the event object.

Interview scenario:

- Explaining why React's synthetic event system prevents browser inconsistencies or how to handle async events in React.

---

53. Explain React reconciliation in detail. How does React decide which components to update? | Virtual DOM / Reconciliation | Advanced

- **Reconciliation** is the process React uses to update the DOM efficiently.
- React compares the **new virtual DOM** with the **previous virtual DOM** (diffing) to determine the minimal set of changes.
- Rules:
- Elements with **different types** are destroyed and recreated.
- Elements with the **same type** are updated with new props and children.
- Lists require **keys** to track items; without unique keys, React may re-render unnecessarily.

Example:

```
const items = ['a', 'b', 'c'].map((item) => <li key={item}>{item}</
```

Pitfalls:

- Using array indices as keys can cause **incorrect updates** when the list changes.

Sample interview scenario:

- Asked to explain why a component re-renders unnecessarily and how keys affect list rendering.

---

54. What is “time slicing” in React Fiber? How does it improve performance? | React Fiber / Performance | Advanced

Explanation:

- Time slicing allows React to **split rendering work into small chunks**, so the main thread isn't blocked for long periods.
- Improves **UI responsiveness**, especially for complex components or large trees.
- React can **pause, yield, and resume work**, making animations and input more fluid.

Example Scenario:

- A component with a heavy loop or multiple nested children may freeze the UI. With time slicing, React updates the UI in increments and keeps the app responsive.

Interview scenario:

- Asked to optimize a React dashboard where large data tables cause janky scrolling.

---

55. What are custom hooks? Can you show an example of when to use one? | Hooks / State Management | Intermediate

Explanation:

- Custom hooks are **reusable functions** that use React hooks internally.
- They encapsulate **stateful logic** and can be shared across multiple components.

Example:

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
```

```
  useEffect(() => {
    async function fetchData() {
      const res = await fetch(url);
      const result = await res.json();
      setData(result);
      setLoading(false);
    }
    fetchData();
  }, [url]);

  return { data, loading };
}

// Usage
function App() {
  const { data, loading } = useFetch('https://api.example.com/data'
  return loading ? <p>Loading...</p> : <pre>{JSON.stringify(data)}<
}
```


Best practices:

- Prefix with `use` to follow React convention.
- Keep hooks focused; each custom hook should handle a single responsibility.

Interview scenario:

- Asked to abstract repeated fetching logic across multiple components in an interview.

---

56. How would you optimize a large React application for performance? | Performance / Optimization | Advanced

Performance optimization in React focuses on **minimizing unnecessary re-renders**, **efficient state management**, and **code splitting**. Key techniques include:

1. **Memoization**

- Use `React.memo` for components and `useMemo` or `useCallback` for values/functions.

2. **Code splitting / lazy loading**

- Dynamically load components with `React.lazy` and `Suspense`.

3. **Avoid anonymous functions in render**

- Functions inside JSX re-create on each render, causing child re-renders.

4. **Virtualization**

- For large lists, use libraries like `react-window` to render only visible items.

5. **Batch state updates**

- Use functional updates in `setState` to prevent multiple re-renders.

Interview Scenario:

- Asked to optimize a dashboard rendering thousands of rows or charts without lag.

---

57. Explain React Suspense for Data Fetching. How is it different from `useEffect`? | Asynchronous Handling / Suspense | Advanced

- `useEffect` fetches data after the component mounts; Suspense allows React to **wait for data before rendering**.

- Suspense lets you **show a fallback UI** while the data is loading.

- Works with libraries like `react-query`, `relay`, or experimental `React.lazy` for data.

Example:

```
import React, { Suspense } from 'react';
const Profile = React.lazy(() => fetchProfile());

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Profile />
    </Suspense>
  );
}
```

Best practices:

- Use Suspense primarily with data-fetching libraries or dynamic imports.
- Keep fallback UI simple for a smooth user experience.

Sample Interview Scenario:

- Asked to implement a loading skeleton for a feed or dashboard.

---

58. What is the difference between client-side routing and server-side routing in React? | Routing / React Router | Intermediate

- **Client-side routing (CSR)**
- Managed in the browser using `react-router`.
- Fast transitions between pages; only the component updates.

- **Server-side routing (SSR)**
- Each URL request hits the server; server returns HTML.
- Better SEO and initial load time; slower navigation without caching.

Example:

```
import { BrowserRouter as Router, Routes, Route } from 'react-route

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Router>
  );
}
```

Interview Scenario:

- Asked why SEO is poor for a SPA and how SSR with Next.js solves it.

---

59. How does `React.StrictMode` help detect unsafe lifecycles and side effects? | Debugging / Strict Mode | Intermediate

- `StrictMode` activates **additional checks and warnings** in development.
- It **does not render any visible UI** or affect production build.
- Checks include:
- Detecting **unsafe lifecycle methods** (`componentWillMount` , etc.).

- Detecting **unexpected side effects** in `useEffect`.
- Warning about **legacy string ref API**.
- Detecting **deprecated findDOMNode usage**.

Example:

```
import React from 'react';

function App() {
  return (
    <React.StrictMode>
      <MyComponent />
    </React.StrictMode>
  );
}
```

Interview Scenario:

- Asked how to identify potential bugs during development without affecting production.

---

60. Explain the difference between `ReactDOM.render` and `ReactDOM.hydrate`. When do you use each? | SSR / Hydration | Advanced

- `ReactDOM.render`
- Renders React components into a **blank container**.
- Used for client-only applications.
- `ReactDOM.hydrate`

- Used when the HTML is **pre-rendered by the server (SSR)**.
- Attaches event listeners and makes HTML interactive without re-rendering it.

Example:

```
// Client-side only
ReactDOM.render(<App />, document.getElementById('root'));

// Server-side rendered
ReactDOM.hydrate(<App />, document.getElementById('root'));
```


Best practices:

- Use `hydrate` only with server-rendered HTML.
- Avoid calling `render` on server-rendered content, as it may replace the HTML and lose SSR benefits.

Interview Scenario:

- Asked to explain how Next.js renders pages on the server and transitions to client-side interactivity.

---

61. What are render props and how do they differ from higher-order components (HOCs)? | Component Patterns / Code Reuse | Intermediate

Render props and HOCs are patterns used to **share logic between components**, but they approach it differently.

Render Props

A render prop is a function prop that a component uses to know what to render. It lets you pass data and behavior to children dynamically.

Example:

```
class MouseTracker extends React.Component {
  state = { x: 0, y: 0 };

  handleMouseMove = (event) => {
    this.setState({ x: event.clientX, y: event.clientY });
  };

  render() {
    return (
      <div onMouseMove={this.handleMouseMove}>
        {this.props.render(this.state)}
      </div>
    );
  }
}

// Usage
function App() {
  return (
    <MouseTracker render={({ x, y }) => (
      <h1>The mouse is at ({x}, {y})</h1>
    )} />
  );
}
```

Higher-Order Components (HOCs)

An HOC is a function that takes a component and returns a new enhanced component.

```
function withMouse(WrappedComponent) {
  return class extends React.Component {
```

```
    state = { x: 0, y: 0 };

    handleMouseMove = (event) => {
      this.setState({ x: event.clientX, y: event.clientY });
    };

    render() {
      return (
        <div onMouseMove={this.handleMouseMove}>
          <WrappedComponent {...this.props} mouse={this.state} />
        </div>
      );
    }
  };
}

function DisplayMouse({ mouse }) {
  return <h1>The mouse is at ({mouse.x}, {mouse.y})</h1>;
}

const EnhancedDisplay = withMouse(DisplayMouse);
```

Differences:

Aspect	Render Props	HOC
How it works	Passes a function as a prop	Wraps a component
Syntax complexity	Slightly simpler and more composable	Requires extra layers
Debugging	Easier due to explicit structure	Can be harder with nested components

Interview Scenario:

- Asked to refactor shared logic between components without repeating code.

- Asked about patterns for code reuse in React.

---

62. Explain the difference between `useEffect` cleanup and `componentWillUnmount` in class components. | Lifecycle / Hooks | Intermediate

Both handle **cleaning up resources**, but they belong to different paradigms.

- `componentWillUnmount` is a lifecycle method used in class components to clean up things like timers or subscriptions before the component is removed.


```
class Timer extends React.Component {
  componentDidMount() {
    this.interval = setInterval(() => console.log('tick'), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return <div>Timer running</div>;
  }
}
```

- `useEffect` with a return function acts similarly in functional components. The cleanup runs when the component unmounts or before the effect runs again.

```
import { useEffect } from 'react';

function Timer() {
  useEffect(() => {
    const interval = setInterval(() => console.log('tick'), 1000);
    return () => clearInterval(interval);
  }, []);
```

```
  return <div>Timer running</div>;
}
```

Key differences:

- `componentWillUnmount` only runs when the component is about to be removed.
- `useEffect` cleanup can run before the next effect, depending on dependencies.

Interview Scenario:

- Asked how to prevent memory leaks in React apps.
- Asked to refactor class-based code to hooks while preserving cleanup logic.

---

63. How would you implement a React component that subscribes to an external data source and cleans up correctly? | Hooks / Asynchronous Updates | Intermediate–Advanced

A component subscribing to an external service (e.g., WebSocket, API polling) must manage subscriptions and clean them up when the component unmounts.

Example: WebSocket Subscription

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

function Chat() {
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    const socket = new WebSocket('wss://chat.example.com');

    socket.onmessage = (event) => {
```

```
      setMessages((prev) => [...prev, event.data]);
    };

    return () => {
      socket.close();
    };
  }, []);

  return (
    <div>
      {messages.map((msg, index) => <div key={index}>{msg}</div>)}
    </div>
  );
}
```

Best practices:

- Always close subscriptions in the cleanup function.
- Avoid stale closures by using functional updates when setting state.
- Use dependency arrays carefully to prevent unwanted resubscriptions.

Interview Scenario:

- Asked to implement real-time data updates without introducing memory leaks or performance issues.

---

64. What is the difference between React server components and client components? | React 18 / SSR | Advanced

React Server Components (RSC) allow parts of the UI to be rendered on the server without sending unnecessary JavaScript to the client, improving performance.

Feature	Server Components	Client Components
Render location	Server-side	Client-side
State & hooks	Limited (no local state/hooks)	Full support
Bundle size	Smaller for client	Larger due to dependencies
Use case	Data fetching, static content	Interactivity, event handlers

Example (conceptual):

```
// Server Component (no interactivity)
export default function UserList({ users }) {
  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}

// Client Component (interactive)
'use client';
import { useState } from 'react';

export default function AddUser({ onAdd }) {
  const [name, setName] = useState('');
  return (
    <div>
      <input value={name} onChange={(e) => setName(e.target.value)}
      <button onClick={() => onAdd(name)}>Add User</button>
    </div>
  );
}
```

Interview Scenario:

- Asked how to optimize initial page load or differentiate interactive vs static parts of the UI in Next.js or similar frameworks.

---

65. How do you prevent unnecessary re-renders in React components? | Performance / Optimization | Advanced

Unnecessary re-renders can slow down applications. Key strategies include:

1. **Use `React.memo`**

- Wrap components to skip re-rendering if props haven't changed.

2. **Use `useMemo` and `useCallback`**

- Memoize values and functions passed as props.

3. **Manage state wisely**

- Avoid storing unrelated data in a shared state that triggers re-renders.

4. **Split state into smaller parts**

- Localize state where possible to limit updates.

5. **Use selectors or context wisely**

- Avoid passing large objects unnecessarily.

Example:

```
const Child = React.memo(({ value }) => {
  console.log('Child rendered');
  return <div>{value}</div>;
});

function Parent() {
```

```
  const [count, setCount] = useState(0);
  const [text, setText] = useState('');

  return (
    <>
      <Child value={count} />
      <input value={text} onChange={(e) => setText(e.target.value)}
      <button onClick={() => setCount(count + 1)}>Increment</buttor
    </>
  );
}
```


Interview Scenario:

- Asked how to optimize a component that re-renders unnecessarily due to state changes elsewhere.

---

66. What is React.memo and when should you use it? | Performance / Optimization | Intermediate–Advanced

`React.memo` is a higher-order component (HOC) that helps prevent unnecessary re-renders by memoizing the result. It shallowly compares the previous and next props and only re-renders the component if the props have changed.

Usage Example:

```
const Child = React.memo(({ name }) => {
  console.log('Child rendered');
  return <div>Hello, {name}!</div>;
});

function Parent() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState("Alice");
```

```
    return (
      <div>
        <Child name={name} />
        <button onClick={() => setCount(count + 1)}>Increment Count</
      </div>
    );
  }
```

In this example, clicking the “Increment Count” button will not cause the `Child` component to re-render because the `name` prop hasn’t changed.

When to use:

- When a component receives props that don’t change frequently.
- For functional components that are expensive to render.

Sample Interview Scenario:

- Asked how to optimize render performance in a large component tree.
- Asked why a child component is re-rendering even though its props seem unchanged.

---

67. What is the difference between `React.memo` and `useMemo`? | Performance / Optimization | Intermediate–Advanced

Both `React.memo` and `useMemo` are used for memoization but in different contexts:

Feature	React.memo	useMemo
What it memoizes	Entire component’s render output	The result of a calculation or function
Where used	Wraps functional components	Inside functional components

Feature	React.memo	useMemo
Purpose	Avoid unnecessary renders	Avoid recalculations

Example using `useMemo`:

```
function Parent({ count }) {
  const expensiveValue = useMemo(() => {
    console.log('Calculating...');
    return count * 2;
  }, [count]);

  return <div>Result: {expensiveValue}</div>;
}
```

When to use:

- Use `React.memo` when you want to avoid unnecessary renders of child components.
- Use `useMemo` when you have expensive computations inside a component.

Interview Scenario:

- Asked how to prevent redundant computations or renders in a performance-sensitive React app.
- Asked to distinguish between component-level and value-level optimizations.

---

68. What is the React event system and how is it different from native DOM events? | Event Handling | Intermediate

React uses a **synthetic event system** that wraps native DOM events to ensure consistent behavior across browsers.

Key differences:

- **Cross-browser compatibility:** React normalizes events to work the same in all browsers.
- **Event delegation:** Events are attached at the root of the document, reducing memory usage and improving performance.
- **Consistent event object:** React's `SyntheticEvent` abstracts away browser-specific quirks.

Example:


```
function Button() {
  const handleClick = (event) => {
    console.log('Button clicked!', event.type);
  };

  return <button onClick={handleClick}>Click Me</button>;
}
```

Common pitfalls:

- Assuming that React's event object persists after the handler is complete (it gets pooled).
- Using browser-specific event properties without checking compatibility.

Best practice:

- If you need to use the event asynchronously, call `event.persist()` or copy the properties you need.

Interview Scenario:

- Asked how React handles events differently from the DOM.

- Asked how to implement event delegation or manage performance in event-heavy components.

---

69. What are custom hooks and why should you use them? | Hooks / Code Reuse | Intermediate

Custom hooks are reusable functions built on top of React's built-in hooks (`useState` , `useEffect` , etc.). They allow you to encapsulate logic that can be shared across multiple components.

Example:

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

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener('resize', handleResize);
    return () => window.removeEventListener('resize', handleResize)
  }, []);

  return width;
}

function Component() {
  const width = useWindowWidth();
  return <div>Window width is {width}px</div>;
}
```

Why use custom hooks:

- Avoid repeating logic across components.

- Improve maintainability and readability.
- Keep component code focused on UI rendering.

Interview Scenario:

- Asked how to share logic between components without duplicating code.
- Asked to refactor a class component with repeated logic into hooks.

---

70. What is time slicing in React and why is it important? | React Internals / Performance | Advanced

Time slicing is a feature introduced with **React Fiber** that allows rendering work to be split into small chunks and spread over multiple frames. This prevents the UI from blocking and keeps it responsive, especially for complex or long tasks.

How it works:

- React prioritizes urgent tasks (like animations or user interactions).
- Non-urgent tasks (like rendering large lists) are spread out to avoid blocking the main thread.

Example use case:
Rendering thousands of items without freezing the UI by using Suspense and lazy loading.

Benefits:

- Smoother animations and interactions.
- Avoids freezing the main thread during heavy rendering.

Limitations:

- Not all work is automatically time sliced; developers need to structure components and use features like `Suspense` to take advantage of it.

Interview Scenario:

- Asked how to handle rendering large datasets without impacting UI responsiveness.
- Asked to explain improvements React Fiber brought over the previous architecture.

---

71. How would you implement optimistic UI updates in React applications? | State Management / UX Patterns | Advanced

Optimistic UI updates refer to updating the UI immediately, assuming that the action (like a server request) will succeed, and rolling back if it fails. This improves perceived performance and responsiveness.

Steps to implement:

1. Update the UI state immediately upon user action.
2. Send the request to the server.
3. If the request succeeds, confirm the update or fetch fresh data.
4. If the request fails, revert the state and show an error.

Example:

```
function TodoApp() {
  const [todos, setTodos] = useState([{ id: 1, text: 'Learn React'
  const [error, setError] = useState(null);

  const addTodo = async (newTodo) => {
    const optimisticTodo = { id: Date.now(), text: newTodo };
```

```
    setTodos([...todos, optimisticTodo]); // Step 1: Optimistic upd

    try {
      await fetch('/api/todos', {
        method: 'POST',
        body: JSON.stringify(optimisticTodo),
      });
    } catch (err) {
      setError('Failed to add todo');
      setTodos(todos); // Rollback
    }
  };

  return (
    <div>
      {error && <p style={{color: 'red'}}>{error}</p>}
      <button onClick={() => addTodo('New Task')}>Add Todo</button>
      <ul>{todos.map(todo => <li key={todo.id}>{todo.text}</li>)}</
    </div>
  );
}
```

When to use:

- When user experience is a priority and server latency can be unpredictable.
- In scenarios like form submissions, data updates, etc.

Interview Scenario:

- Asked how to make the UI feel snappy even if the backend is slow.
- Asked to explain how to handle failure cases in real-time applications.

---

72. How do you handle accessibility (a11y) in a React application? | UI / Accessibility | Intermediate

Accessibility ensures that your application can be used by people with disabilities. React encourages semantic HTML and attributes like `aria-*` to improve accessibility.

Key techniques:

- Use semantic elements like `<button>`, `<nav>`, `<form>`.
- Add `aria-label`, `aria-describedby`, and `aria-expanded` where needed.
- Ensure proper keyboard navigation (using `tabIndex`, etc.).
- Manage focus for modals or dialogs.
- Use libraries like `react-aria` or `react-axe` to audit accessibility.

Example:

```
function Modal({ isOpen, onClose }) {
  return isOpen ? (
    <div role="dialog" aria-modal="true" aria-labelledby="dialog-ti
      <h2 id="dialog-title">Subscribe</h2>
      <button onClick={onClose}>Close</button>
    </div>
  ) : null;
}
```

Interview Scenario:

- Asked how to make web applications inclusive.
- Asked how to fix keyboard navigation issues.
- Asked how to use `aria` attributes in forms or dialogs.

73. What are React Portals and when would you use them? Can you give a complex example? | UI / DOM Manipulation | Intermediate

React Portals allow rendering components outside of the parent DOM hierarchy while keeping React's state and context intact. This is useful for modals, tooltips, dropdowns, etc., that need to break out of layout constraints.

Example:

```
import ReactDOM from 'react-dom';

function Modal({ children, isOpen, onClose }) {
  if (!isOpen) return null;

  return ReactDOM.createPortal(
    <div className="modal-backdrop" onClick={onClose}>
      <div className="modal-content" onClick={(e) => e.stopPropagat
        {children}
      </div>
    </div>,
    document.getElementById('modal-root')
  );
}
```

Complex example:

- Modal with focus trap and keyboard accessibility.
- Tooltips rendered relative to a button but outside overflow boundaries.

When to use:

- UI elements that need to overlay other content.

- Avoid issues with CSS stacking context, z-index, or overflow.

Interview Scenario:

- Asked how to implement modals or dropdowns without layout issues.
- Asked how to separate overlay elements while maintaining state and props.

---

74. Explain the difference between shallow rendering and full DOM rendering in React testing. | Testing / React Testing Library | Intermediate

In testing, shallow rendering and full DOM rendering are two different approaches to isolate component behavior.

Shallow rendering:

- Renders the component without rendering its child components.
- Used to test a component's output in isolation.
- Provided by tools like `enzyme` .

Full DOM rendering:

- Renders the entire component tree.
- Suitable for integration tests where you need to interact with nested components or lifecycle methods.

Example:

```
import { shallow, mount } from 'enzyme';
import MyComponent from './MyComponent';
```

```
test('shallow test', () => {
  const wrapper = shallow(<MyComponent />);
  expect(wrapper.find('button').exists()).toBe(true);
});

test('full render test', () => {
  const wrapper = mount(<MyComponent />);
  expect(wrapper.find('ChildComponent').length).toBe(1);
});
```

When to use:

- Use shallow rendering for unit tests.
- Use full DOM rendering for complex interactions or state changes.

Interview Scenario:

- Asked how to write test cases that isolate logic versus testing full UI behavior.
- Asked when to use enzyme vs React Testing Library.

---

75. What is the difference between server components and client components in React? | React 18+ / Architecture | Advanced

React 18 introduces **server components**, which are rendered on the server and streamed to the client, unlike client components that are fully loaded and executed on the browser.

Key differences:

Feature	Server Components	Client Components
Rendering location	Server	Browser
JavaScript payload	Minimal or none	Full bundle
Data fetching	Direct from server-side APIs	Needs client-side calls
Use case	Static content, SEO, performance	Interactivity, event handling

Example:


```
// Server Component (no hooks or browser-specific code)
export default function ProductList({ products }) {
  return (
    <div>
      {products.map(product => <div key={product.id}>{product.name}
    </div>
  );
}

// Client Component (with interactivity)
"use client";
import { useState } from 'react';

export default function CartButton() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>Add {count}</b
}
```

When to use:

- Server components: for static or SEO-heavy pages.
- Client components: for interactive or state-driven UI.

Interview Scenario:

- Asked how React's architecture is evolving.
- Asked how server-side rendering impacts performance and scalability.

akshay

KEEP READING

Effective Error Handling Patterns in Node.js

By Tushar Kulkarni

Git-Flow & Trunk-Based Development

By Himanshi Rana

asyncio & await

By Pranav Shah

Insert into a BST

By devangini123

Guess Higher or Lower

By akshay

Valid Palindrome | Approach 1 Extra Space

By akshay

ADD A COMMENT

Search

Search

RECENT POSTS

[Kahn's Algorithm Topological Sort BFS](#)

November 27, 2025

[Topological Sort – DFS](#)

November 26, 2025

[Detect Cycle in Undirected Connected Graph](#)

November 26, 2025

[Reconstruct Itinerary](#)

November 24, 2025

LATEST POSTS

CONNECT WITH US!

Subscribe to Stay Updated

Stay ahead in the world of tech with our exclusive newsletter! Subscribe now for regular updates on the latest trends, valuable coding resources, and tips to boost your frontend development skills.

Email

SUBSCRIBE

UPSKILL YOURSELF

Explore

Introduction to Greedy Algorithm

September 11, 2025 — Updated: September 11, 2025 — ALGORITHMS No Comments 2 Mins Read —

By devangini123

Greedy Algorithm

- Greedy is a problem solving approach where we make `locally optimal choices` , with a hope that it will lead to `global optimal solution` .

Take best choice at the moment, without worrying about the future.

Examples

Example 1: `Coin Change Problem`

Given an amount of n rupees and an unlimited supply of coins or notes of denominations {1, 2, 5, 10}. we have to find the minimum number of coins required to make up the 18 amount.

Intuition:

- Firstly, **choose the maximum denomination coin**. Suppose we choose `10`, now `8` is remaining.
- If we choose the `maximum coin (10)` again, it exceeds the target, so we choose the **second largest denomination**, which is `5`.
- Now, `3` **amount is left**, so we choose `2`.
- Finally, we choose `1`. Now our target is **complete**.

Example 2: Knap Sack Problem: `Fractional Knap Sack Problem`

Given n items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Value = {60, 100, 120}, Weight = {10, 20, 30}, Weight only carry = 50kg

Intuition:

- Since the **bag has limited capacity**, we cannot take all items, so we must `choose` them wisely.
- Instead of directly picking the **highest value items**, we should compare items based on their `value-to-weight` ratio (value per unit weight).
- This way, we prioritize items that give the `maximum profit for the minimum weight`.
- By filling the bag with items having the **highest value/weight ratio first**, we `maximize` the total profit within the capacity constraint.

Types of Problems:

- `Optimization Problems` **(min/max result)**
- **Activity Selection / Interval Scheduling**
- **Fractional Knapsack**
- **Job Sequencing with deadlines**

- `Graph Problems`
- **MST, Kruskal's / Prim's Algorithm**
- **Shortest Path**

- `Scheduling Problems`
- **CPU Scheduling (Shortest first)**

- `Game / Puzzles and more`

Greedy Problem Hints

- **Maximum number of**
- **Minimum Cost / Maximum Profit**
- **Schedule / allocate / assign efficiently**
- **Shortest / Smallest / Largest / Longest (with constraints)**

- **Non-overlapping intervals**

devangini123

KEEP READING

Kahn's Algorithm Topological Sort BFS

By devangini123

Topological Sort – DFS

By devangini123

Detect Cycle in Undirected Connected Graph

By devangini123

Reconstruct Itinerary

By devangini123

All Paths from Source to Target

By devangini123

DFS Recursive

By devangini123

ADD A COMMENT

Search

Search

RECENT POSTS

[Kahn's Algorithm Topological Sort BFS]

November 27, 2025

[Topological Sort – DFS]

November 26, 2025

[Detect Cycle in Undirected Connected Graph]

November 26, 2025

[Reconstruct Itinerary]

November 24, 2025

LATEST POSTS

CONNECT WITH US!

Subscribe to Stay Updated

Stay ahead in the world of tech with our exclusive newsletter! Subscribe now for regular updates on the latest trends, valuable coding resources, and tips to boost your frontend development skills.

Email

SUBSCRIBE

Best Time to Buy and Sell Stock II

September 11, 2025 — Updated: September 11, 2025 — ALGORITHMS No Comments 2 Mins Read —

By devangini123

Problem Statement:

You are given an integer array `prices` where `prices[i]` is the price of a given stock on the `i`th day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the **maximum** profit you can achieve*.

Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 7

Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4. Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3. Total profit is 4 + 3 = 7.

Example 2:

Input: prices = [1,2,3,4,5]

Output: 4

Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4. Total profit is 4.

Example 3:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

Constraints

- `1 <= prices.length <= 3 * 10^4`
- `0 <= prices[i] <= 10^4`

Approach:

- If the price today is higher than yesterday, we add the profit `(prices[i] - prices[i-1])` to our **total**.
- Otherwise, we skip (no loss is counted). This way, we **sum all upward moves to get the maximum profit.**

Time & Space Complexity:

Time Complexity: `O(n)`

Space Complexity: `O(1)`

Dry Run

Input: `prices = [7, 1, 5, 3, 6, 4]`

```
 Step 1: Initialize ans = 0 Step 2: Iterate over prices i = 1 → profit = prices[1] - prices[0] = 1 - 7 = -6 profit ≤ 0 → no change → ans = 0 i = 2 → profit = prices[2] - prices[1] = 5 - 1 = 4 profit > 0 → ans = 0 + 4 = 4 i = 3 → profit = prices[3] - prices[2] = 3 - 5 = -2 profit ≤ 0 → no change → ans = 4 i = 4 → profit = prices[4] - prices[3] = 6 - 3 = 3 profit > 0 → ans = 4 + 3 = 7 i = 5 → profit = prices[5] - prices[4] = 4 - 6 = -2 profit ≤ 0 → no change → ans = 7 Step 3: End Return ans = 7
```

Output: `Result: 7`

JavaScript | Python | Java | C++ | C | C#

```
var maxProfit = function(prices) {
    let ans = 0;
    for(let i=1; i 0){
            ans += profit;
        }
    }
    return ans;
};
```

devangini123

KEEP READING

Kahn's Algorithm Topological Sort BFS

By devangini123

Topological Sort – DFS

By devangini123

Detect Cycle in Undirected Connected Graph

By devangini123

Reconstruct Itinerary

By devangini123

All Paths from Source to Target

By devangini123

DFS Recursive

By devangini123

ADD A COMMENT

Search

Search

RECENT POSTS

Kahn's Algorithm Topological Sort BFS

November 27, 2025

Topological Sort – DFS

November 26, 2025

[Detect Cycle in Undirected Connected Graph]

November 26, 2025

[Reconstruct Itinerary]

November 24, 2025

LATEST POSTS

CONNECT WITH US!

Subscribe to Stay Updated

Stay ahead in the world of tech with our exclusive newsletter! Subscribe now for regular updates on the latest trends, valuable coding resources, and tips to boost your frontend development skills.

Email

SUBSCRIBE

UPSKILL YOURSELF

Explore

Lemonade Change

September 11, 2025 — Updated: September 11, 2025 — ALGORITHMS No Comments 3 Mins Read —

By devangini123

Problem Statement:

At a lemonade stand, each lemonade costs `$5`. Customers are standing in a queue to buy from you and order one at a time (in the order specified by bills). Each customer will only buy one lemonade and pay with either a `$5`, `$10`, or `$20` bill. You must provide the correct change to each customer so that the net transaction is that the customer pays `$5`.

Note that you do not have any change in hand at first.

Given an integer array `bills` where `bills[i]` is the bill the `i`th customer pays, return `true` *if you can provide every customer with the correct change, or* `false` *otherwise.*

Example 1:

Input: bills = [5,5,5,10,20]

Output: true

Explanation:

- **From the first 3 customers, we collect three $5 bills in order.**
- **From the fourth customer, we collect a $10 bill and give back a $5.**
- **From the fifth customer, we give a $10 bill and a $5 bill.**
- **Since all customers got correct change, we output true.**

Example 2:

Input: bills = [5,5,10,10,20]

Output: false

Explanation:

- **From the first two customers in order, we collect two $5 bills.**
- **For the next two customers in order, we collect a $10 bill and give back a $5 bill.**
- **For the last customer, we can not give the change of $15 back because we only have two $10 bills.**
- **Since not every customer received the correct change, the answer is false.**

Constraints

- `1 <= bills.length <= 10^5`
- `bills[i] is either 5, 10, or 20.`

Approach:

- Maintain a **wallet** to track count of `$5` and `$10` bills.
- **Iterate** over each customer's bill:
- If bill is `$5` → add to wallet.
- If bill is `$10` → give one `$5` as change and add one `$10` .
- If bill is `$20` →

- Prefer giving one `$10` + one `$5` if possible.
- Otherwise, give three `$5`.

- After each transaction, if wallet runs out of required bills (negative count), return `false`.
- If all **transactions succeed**, **return** `true`.

Time & Space Complexity:

Time Complexity: `O(n)`

Space Complexity: `O(1)`

Dry Run

Input: `bills = [5, 5, 5, 10, 20]`

```
Step 1: Initialize
wallet = [0, 0] // [count of $5, count of $10]
Step 2: Process each bill
i = 0 → bill = 5
Add one $5 → wallet = [1, 0]
i = 1 → bill = 5
Add one $5 → wallet = [2, 0]
i = 2 → bill = 5
Add one $5 → wallet = [3, 0]
i = 3 → bill = 10
Give back one $5, keep one $10
wallet = [2, 1]
i = 4 → bill = 20
Prefer $10 + $5 if possible
Use one $10 and one $5 → wallet = [1, 0]
Step 3: Validation
At each step, wallet[0] (number of $5) never goes negative.
Step 4: End
Return true
```

> **Output:** `Result: true`

JavaScript | Python | Java | C++ | C | C#

```
var lemonadeChange = function(bills) {
    let wallet = [0, 0];
    for (let i = 0; i < bills.length; ++i) {
        if (bills[i] === 5) {
            wallet[0]++;
        } else if (bills[i] === 10) {
            wallet[1]++;
            wallet[0]--;
        } else { // bill is 20
            if (wallet[1] > 0) {
                wallet[1]--;
                wallet[0]--;
            } else {
                wallet[0] -= 3;
            }
        }
        if (wallet[0] < 0) {
            return false;
        }
    }
    return true;
};
```

---

devangini123

KEEP READING

Kahn's Algorithm Topological Sort BFS

By devangini123

Topological Sort – DFS

By devangini123

Detect Cycle in Undirected Connected Graph

By devangini123

Reconstruct Itinerary

By devangini123

All Paths from Source to Target

By devangini123

DFS Recursive

By devangini123

ADD A COMMENT

Search

Search

RECENT POSTS

Kahn's Algorithm Topological Sort BFS

November 27, 2025

Topological Sort – DFS

November 26, 2025

Detect Cycle in Undirected Connected Graph

November 26, 2025

[Reconstruct Itinerary]

November 24, 2025

LATEST POSTS

CONNECT WITH US!

Subscribe to Stay Updated

Stay ahead in the world of tech with our exclusive newsletter! Subscribe now for regular updates on the latest trends, valuable coding resources, and tips to boost your frontend development skills.

Email

SUBSCRIBE

UPSKILL YOURSELF

Explore

Assign Cookies

September 11, 2025 — Updated: September 11, 2025 — ALGORITHMS AND DATA STRUCTURES No Comments

3 Mins Read — By devangini123

Problem Statement:

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.

Each child `i` has a greed factor `g[i]`, which is the minimum size of a cookie that the child will be content with; and each cookie `j` has a size s[j]. If `s[j] >= g[i]`, we can assign the cookie `j` to the child `i`, and the child `i` will be content. Your goal is to maximize the number of your content children and output the maximum number.

Example 1:

Input: g = [1,2,3], s = [1,1]

Output: 1

Explanation: You have 3 children and 2 cookies. The greed factors of 3 children are 1, 2, 3. And even though you have 2 cookies, since their size is both 1, you could only make the child whose greed factor is 1 content. You need to output 1.

Example 2:

Input: g = [1,2], s = [1,2,3]

Output: 2

Explanation: You have 2 children and 3 cookies. The greed factors of 2 children are 1, 2. You have 3 cookies and their sizes are big enough to gratify all of the children, You need to output 2.

Constraints

- `1 <= g.length <= 3 * 10^4`
- `0 <= s.length <= 3 * 10^4`
- `1 <= g[i], s[j] <= 2^31 - 1`

Approach:

- **Sort** both arrays `g` and `s` in **ascending order**.
- **Use two pointers**
- `i` → tracks children
- `j` → tracks cookies
- **Traverse** through cookies (`s`):
- If current cookie satisfies the child's greed (`s[j] >= g[i]`), assign it → move to next child (`i++`).
- Always move to **next** cookie `(j++)` .
- **Return** `i` → number of children satisfied.

Time & Space Complexity:

Time Complexity: `O(n log n + m log m)`

Space Complexity: `O(1)`

Dry Run

Input: `g = [1,2,3], s = [1,1]`

```
Step 1: Sort both arrays
g = [1, 2, 3]
s = [1, 1]

Step 2: Initialize
i = 0 (index for g → children)
j = 0 (index for s → cookies)

Step 3: Start while loop (j < s.length)
j = 0 → s[0] = 1, g[0] = 1
Since 1 >= 1 → child gets cookie → i = 1
j = 1 → s[1] = 1, g[1] = 2
Since 1 < 2 → child not satisfied
Increment j = 2 (loop ends)

Step 4: End
i = 1 (number of content children)
```

Output: `Result: 1`

JavaScript | Python | Java | C++ | C | C#

```
var findContentChildren = function(g, s) {
    s.sort((a, b) => a - b);
```

```
    g.sort((a, b) => a - b);

    let i = 0; // child index
    let j = 0; // cookie index

    while (i < g.length && j < s.length) {
        if (s[j] >= g[i]) {
            ++i;
            ++j;
        } else {
            ++j;
        }
    }
    return i;
};
```


devangini123

KEEP READING

Kahn's Algorithm Topological Sort BFS

By devangini123

Topological Sort – DFS

By devangini123

Detect Cycle in Undirected Connected Graph

By devangini123

Reconstruct Itinerary

By devangini123

All Paths from Source to Target

DFS Recursive

By devangini123

By devangini123

ADD A COMMENT

Search

Search

RECENT POSTS

Kahn's Algorithm Topological Sort BFS

November 27, 2025

Topological Sort – DFS

November 26, 2025

Detect Cycle in Undirected Connected Graph

November 26, 2025

Reconstruct Itinerary

November 24, 2025

LATEST POSTS

CONNECT WITH US!

Subscribe to Stay Updated

Stay ahead in the world of tech with our exclusive newsletter! Subscribe now for regular updates on the latest trends, valuable coding resources, and tips to boost your frontend development skills.

Email

SUBSCRIBE

UPSKILL YOURSELF

Explore

NamasteDev.com

COURSES

Namaste DSA

Namaste React

Namaste Frontend System Desgin

Namaste Javascript

Crack Frontend Interview

COMMUNITY

YouTube

LinkedIn

Discord

Instagram

Twitter

Namastedev Platform

CONTACT US

Support@namastedev.com

Subscribe to Stay Updated

Stay ahead in the world of tech with our exclusive newsletter! Subscribe now for regular updates on the latest trends, valuable coding resources, and tips to boost your frontend development skills.

Email

SUBSCRIBE