100+ React Technical Interview Problems with Solutions

1. React Basics

Q1. Create a functional component that displays "Hello World".

```
function HelloWorld() {
  return <h1>Hello World</h1>;
}
export default HelloWorld;
```

Q2. Class component that accepts name prop and displays "Hello {name}".

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

class Greeting extends Component {
  render() {
    return <h1>Hello {this.props.name}</h1>;
  }
}
export default Greeting;
```

Q3. Convert the above class component to a functional component.

```
function Greeting({ name }) {
  return <h1>Hello {name}</h1>;
}
```

Q4. Render a list of items using .map().

```
const fruits = ["Apple", "Banana", "Orange"];
function FruitList() {
  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>{fruit}</li>
      ))}
    </ul>
  );
}
```

Q5. Render a component conditionally.

```
function User({ isLoggedIn }) {
  return <h1>{isLoggedIn ? "Logged In" : "Guest"}</h1>;
}
```

Q6. Create a button that shows an alert on click.

```
function AlertButton() {
  const handleClick = () => alert("Button clicked");
  return <button onClick={handleClick}>Click Me</button>;
}
```

Q7. Display current date and time using state.

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

function Clock() {
  const [time, setTime] = useState(new Date().toLocaleTimeString());
  useEffect(() => {
    const interval = setInterval(() => setTime(new Date().toLocaleTimeString()), 1000);
    return () => clearInterval(interval);
  }, []);
  return <h2>{time}</h2>;
}
```

Q8. Pass data from parent to child using props.

```
function Parent() {
  const name = "Krishnendu";
  return <Child name={name} />;
}
function Child({ name }) {
  return <p>Hello {name}</p>;
}
```

Q9. Render multiple components dynamically from an array of objects.

```
const users = [{id:1,name:"A"},{id:2,name:"B"}];
function UserList() {
  return (
    <div>
      {users.map(u => <p key={u.id}>{u.name}</p>)}
    </div>
  );
}
```

Q10. Explain difference between functional and class components.
Answer: Functional: simple, hooks-based. Class: uses this.state and lifecycle methods.

2. State & Event Handling

Q11. Counter with increment and decrement.

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
    </>
  );
}
```

Q12. Input field updates live text.

```
function InputDemo() {
  const [text, setText] = useState("");
  return (
    <>
      <input onChange={e => setText(e.target.value)} />
      <p>{text}</p>
    </>
  );
}
```

Q13. Toggle visibility of a div.

```
function ToggleDiv() {
  const [visible, setVisible] = useState(true);
  return (
    <>
      <button onClick={() => setVisible(!visible)}>Toggle</button>
      {visible && <div>Now you see me!</div>}
    </>
  );
}
```

Q14. Pass arguments to event handler.

```
function List({ items }) {
  const handleClick = item => alert(item);
  return items.map(i => <button key={i} onClick={() => handleClick(i)}>{i}</button>);
}
```

Q15. Form with controlled input.

```
function LoginForm() {
  const [form, setForm] = useState({username:""});
  return (
    <input value={form.username} onChange={e => setForm({username:e.target.value})}/>
  );
}
```

Q16. Multiple controlled inputs with one handler.

```
function MultiInputForm() {
  const [form, setForm] = useState({name:"",email:""});
  const handleChange = e => setForm({...form,[e.target.name]:e.target.value});
  return (
    <>
      <input name="name" value={form.name} onChange={handleChange}/>
      <input name="email" value={form.email} onChange={handleChange}/>
    </>
  );
}
```

Q17. Submit form with validation.

```
const handleSubmit = e => {
  e.preventDefault();
  if(form.name.length < 3) alert("Name too short");
};
```

Q18. Increment using previous state.

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

Q19. Handle checkbox input.

```
function CheckboxDemo() {
  const [checked, setChecked] = useState(false);
  return <input type="checkbox" checked={checked} onChange={e=>setChecked(e.target.checked)}/>;
}
```

Q20. Radio button selection.

```
function RadioDemo() {
  const [choice, setChoice] = useState("A");
  return <>
<input type="radio" value="A" checked={choice==="A"} onChange={e=>setChoice(e.target.value)}/>A
<input type="radio" value="B" checked={choice==="B"} onChange={e=>setChoice(e.target.value)}/>B
  </>;
}
```

3. Lifecycle & useEffect

Q21. componentDidMount example (class).

```
componentDidMount() {
  console.log("Mounted");
}
```

Q22. useEffect equivalent (functional).

```
useEffect(() => { console.log("Mounted"); }, []);
```

Q23. Cleanup with useEffect.

```
useEffect(() => {
  const timer = setInterval(()=>console.log("tick"),1000);
  return () => clearInterval(timer);
}, []);
```

Q24. Fetch API data on mount.

```
useEffect(() => {
  fetch("url").then(r=>r.json()).then(setData);
}, []);
```

Q25. Update document title with state.

```
useEffect(() => { document.title = `Count: ${count}`; }, [count]);
```

Q26. componentDidUpdate equivalent.

```
useEffect(() => { console.log("Count changed", count); }, [count]);
```

Q27. componentWillUnmount example.

```
useEffect(() => {
  return () => console.log("Cleanup before unmount");
}, []);
```

Q28. Fetch data when prop changes.

```
useEffect(() => { fetchData(id); }, [id]);
```

Q29. Multiple useEffects for separation.

```
useEffect(() => {}, []); // mount
useEffect(() => {}, [state]); // update
```

Q30. useLayoutEffect example.

```
useLayoutEffect(() => { console.log("Runs before paint"); }, []);
```

4. Lists & Keys

Q31. List rendering with unique keys.

```
{users.map(u => <p key={u.id}>{u.name}</p>)}
```

Q32. Nested list rendering.

```
categories.map(c=>(
  <div key={c.id}>
    <h2>{c.name}</h2>
    <ul>{c.items.map(i=><li key={i}>{i}</li>)}</ul>
  </div>
))
```

Q33. Conditional list rendering.

```
{items.length>0 ? items.map(...) : <p>No items</p>}
```

Q34. Filter list dynamically.

```
items.filter(i=>i.includes(search)).map(...)
```

Q35. Sort list dynamically.

```
items.sort((a,b)=>a-b).map(...)
```

Q36. Map object keys to list.

```
Object.keys(obj).map(key=><p key={key}>{obj[key]}</p>)
```

Q37. Nested components in a list.

```
items.map(i=> <Item key={i.id} data={i}/>)
```

Q38. Memoized list item to prevent re-render.

```
const ListItem = React.memo(({data})=><li>{data}</li>);
```

Q39. Dynamic CSS classes in list items.

```
<li className={item.active ? "active":"inactive"}>{item.name}</li>
```

Q40. List with delete button.

```
function ListWithDelete() {
  const [items,setItems]=useState(["A","B"]);
  const remove=i=>setItems(items.filter(x=>x!==i));
  return items.map(i=> <li key={i}>{i}<button onClick={()=>remove(i)}>Delete</button></li>)
}
```

5. Context API

Q41. Create a theme context.

```
const ThemeContext = React.createContext("light");
```

Q42. Provide context to components.

```
<ThemeContext.Provider value="dark">
  <Toolbar />
</ThemeContext.Provider>
```

Q43. Consume context using useContext.

```
const theme = useContext(ThemeContext);
```

Q44. Class component context consumption.

```
static contextType = ThemeContext;
```

Q45. Nested context consumption.

```
const theme = useContext(ThemeContext);
const user = useContext(UserContext);
```

6. React Router

Q46. Basic route setup.

```
<Routes>
  <Route path="/" element={<Home/>}/>
  <Route path="/about" element={<About/>}/>
</Routes>
```

Q47. Link between pages.

```
<Link to="/about">About</Link>
```

Q48. Nested routes.

```
<Route path="dashboard" element={<Dashboard/>}>
  <Route path="settings" element={<Settings/>}/>
</Route>
```

Q49. Redirect to another page.

```
<Navigate to="/login" replace />
```

Q50. Dynamic route parameter.

```
<Route path="/user/:id" element={<User/>}/>
```

7. Redux & State Management

Q51. Basic store setup.

```
import { createStore } from 'redux';
const reducer = (state={count:0}, action)=>action.type==='INCREMENT'?{count:state.count+1}:state;
const store = createStore(reducer);
```

Q52. Dispatch an action.

```
store.dispatch({type:'INCREMENT'});
```

Q53. Subscribe to store.

```
store.subscribe(()=>console.log(store.getState()));
```

Q54. Combine reducers.

```
combineReducers({user:userReducer,count:countReducer});
```

Q55. Connect React component to Redux store.

```
const mapStateToProps=state=>({count:state.count});
export default connect(mapStateToProps)(Counter);
```

8. Advanced Hooks

Q56. useReducer for counter.

```
const reducer = (state, action) => {
  switch(action.type){
    case 'increment': return {count: state.count+1};
    case 'decrement': return {count: state.count-1};
    default: return state;
  }
};
function Counter() {
  const [state, dispatch] = useReducer(reducer, {count:0});
  return <>
    <button onClick={()=>dispatch({type:'decrement'})}>-</button>
    {state.count}
    <button onClick={()=>dispatch({type:'increment'})}>+</button>
  </>;
}
```

Q57. useCallback to memoize function.

```
const handleClick = useCallback(()=>alert("Clicked"), []);
```

Q58. useMemo to memoize expensive calculation.

```
const result = useMemo(()=> expensiveCalc(data), [data]);
```

Q59. Custom hook to fetch API data.

```
function useFetch(url){
  const [data, setData] = useState(null);
  useEffect(()=>{ fetch(url).then(res=>res.json()).then(setData); }, [url]);
  return data;
}
```

Q60. Track previous value with useRef.

```
const prevCount = useRef();
useEffect(()=>{ prevCount.current = count }, [count]);
```

Q61. Focus input on mount using useRef.

```
const inputRef = useRef();
useEffect(()=>{ inputRef.current.focus() }, []);
<input ref={inputRef}/>
```

Q62. useImperativeHandle example.

```
const FancyInput = forwardRef((props, ref) => {
  const inputRef = useRef();
  useImperativeHandle(ref, ()=>({ focus: ()=>inputRef.current.focus() }));
  return <input ref={inputRef}/>;
});
```

Q63. useLayoutEffect for measuring DOM before paint.

```
useLayoutEffect(()=>{ console.log(divRef.current.offsetHeight); }, []);
```

Q64. useDebugValue to debug custom hooks.

```
useDebugValue(count > 5 ? "High" : "Low");
```

Q65. Lazy initialization in useState.

```
const [state, setState] = useState(()=> computeExpensiveValue());
```

9. Higher-Order Components (HOCs)

Q66. Basic HOC to log props.

```
function withLogger(Component){
  return function(props){
    console.log(props);
    return <Component {...props}/>;
  }
}
```

Q67. HOC to add loading spinner.

```
function withLoading(Component){
  return ({loading, ...props}) => loading ? <p>Loading...</p> : <Component {...props}/>;
}
```

Q68. HOC for authentication.

```
function withAuth(Component){
  return props => currentUser ? <Component {...props}/> : <Navigate to="/login"/>;
}
```

Q69. HOC to inject theme prop.

```
function withTheme(Component){
  return props => <Component {...props} theme="dark"/>;
}
```

Q70. HOC to track component mount/unmount.

```
function withTracker(Component){
  return props => {
    useEffect(()=>{ console.log("Mounted"); return ()=>console.log("Unmounted") }, []);
    return <Component {...props}/>;
  }
}
```

10. Error Boundaries

Q71. Class-based error boundary.

```
class ErrorBoundary extends React.Component {
  state = { hasError:false };
  static getDerivedStateFromError() { return {hasError:true}; }
  componentDidCatch(error, info) { console.log(error, info); }
  render() { return this.state.hasError ? <h1>Error occurred</h1> : this.props.children; }
}
```

Q72. Wrap component with error boundary.

```
<ErrorBoundary><MyComponent/></ErrorBoundary>
```

Q73. Reset error state on retry.

```
<button onClick={()=>this.setState({hasError:false})}>Retry</button>
```

11. React Suspense & Lazy Loading

Q74. Lazy load a component.

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

Q75. Use Suspense to show fallback.

```
<Suspense fallback={<p>Loading...</p>}><LazyComponent/></Suspense>
```

Q76. Multiple lazy-loaded components.

```
<Suspense fallback={<p>Loading...</p>}>
  <LazyComp1/>
  <LazyComp2/>
</Suspense>
```

12. Portals

Q77. Render modal using portal.

```
ReactDOM.createPortal(<div>Modal</div>, document.getElementById('modal-root'));
```

Q78. Dynamic portal content.

```
const Portal = ({children})=>ReactDOM.createPortal(children,
document.getElementById('portal-root'));
```

13. Forms & Validation

Q79. Form validation with required fields.

```
if(!form.username) alert("Username required");
```

Q80. Show error messages dynamically.

```
<p style={{color:'red'}}>{errors.username}</p>
```

Q81. Controlled select input.

```
<select value={choice} onChange={e=>setChoice(e.target.value)}><option>A</option>
<option>B</option></select>
```

Q82. Dynamic form fields.

```
fields.map(f=><input key={f.id} name={f.name} value={f.value} onChange={handleChange}/>)
```

Q83. Form reset after submit.

```
setForm({username:'',password:''});
```

14. Performance Optimization

Q84. React.memo for functional component.

```
const MemoComp = React.memo(({value})=> <p>{value}</p>);
```

Q85. useMemo to memoize computation.

```
const total = useMemo(()=>arr.reduce((a,b)=>a+b,0), [arr]);
```

Q86. useCallback for event handler.

```
const handleClick = useCallback(()=>console.log("clicked"), []);
```

Q87. Lazy load components to improve performance.

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

Q88. Avoid unnecessary re-renders in parent-child.

```
<Child value={value} onClick={handleClick}/>
```

15. Testing with Jest & React Testing Library

Q89. Test component renders.

```
render(<MyComp/>); expect(screen.getByText("Hello")).toBeInTheDocument();
```

Q90. Test button click event.

```
fireEvent.click(screen.getByText("Click"));
```

Q91. Test input change.

```
fireEvent.change(screen.getByRole("textbox"), {target:{value:"Hi"}});
```

Q92. Test conditional rendering.

```
expect(screen.queryByText("Logged In")).toBeNull();
```

Q93. Mock API fetch call.

```
global.fetch = jest.fn(()=>Promise.resolve({json:()=>Promise.resolve(data)}));
```

16. Advanced Patterns & Real-World Problems

Q94. Infinite scrolling list.

```
useEffect(()=>{ window.addEventListener("scroll", handleScroll);
return ()=>window.removeEventListener("scroll", handleScroll); }, []);
```

Q95. Debounce input change.

```
const debounced = useDebounce(value, 300);
```

Q96. Drag-and-drop list.

```
import React, { useState } from "react";
import { DragDropContext, Droppable, Draggable } from "react-beautiful-dnd";

const initialItems = [
  { id: "1", content: "Item 1" },
  { id: "2", content: "Item 2" },
  { id: "3", content: "Item 3" },
];

function DragDropList() {
  const [items, setItems] = useState(initialItems);
```

```
// Reorder items after drag
const handleOnDragEnd = (result) => {
  if (!result.destination) return; // dropped outside list
  const newItems = Array.from(items);
  const [reorderedItem] = newItems.splice(result.source.index, 1);
  newItems.splice(result.destination.index, 0, reorderedItem);
  setItems(newItems);
};
```

```
return (
  <DragDropContext onDragEnd={handleOnDragEnd}>
    <Droppable droppableId="droppable">
      {(provided) => (
        <ul {...provided.droppableProps} ref={provided.innerRef}>
          {items.map((item, index) => (
            <Draggable key={item.id} draggableId={item.id} index={index}>
              {(provided) => (
                <li
                  ref={provided.innerRef}
                  {...provided.draggableProps}
                  {...provided.dragHandleProps}
                  style={{
                    padding: "8px",
                    margin: "4px 0",
                    backgroundColor: "#f0f0f0",
                    border: "1px solid #ccc",
```

```
                      ...provided.draggableProps.style,
                    }}
                  >
                    {item.content}
                  </li>
                )}
              </Draggable>
            ))}
            {provided.placeholder}
          </ul>
        )}
      </Droppable>
    </DragDropContext>
  );
}

export default DragDropList;
```

Q97. Dark mode toggle using context.

```
const ThemeContext = createContext(); <ThemeContext.Provider value={theme}>
```

Q98. Multi-step form.

```
const [step,setStep]=useState(1);
```

Q99. Image lazy loading.

```
<img loading="lazy" src="image.jpg"/>
```

Q100. Dynamic tab component.

```
tabs.map(t=> <button onClick={()=>setActive(t.id)}>{t.name}</button>)
```

Q101. Toast notifications using context.

```
<ToastProvider><App/></ToastProvider>
```