DAY-21 MORNING ASSESSENT

**Code splitting by route**

1.Code splitting is breaking your bundle into smaller chunks that load on demand. It reduces initial load time, improves performance, and loads only what’s needed.

2.Route-based code splitting uses React.lazy() and Suspense.

Example:  
  
const Home = lazy(() => import("./Home"));  
<Suspense fallback={<div>Loading...</div>}><Route path="/" element={<Home />} /></Suspense>

3.The fallback prop in Suspense shows a temporary UI (e.g., spinner, text) while loading a lazy component.

4.It improves load time by downloading only the code required for the initial route instead of the whole app bundle.

5.Route splitting splits at page level (good for large apps), while component splitting splits individual heavy components (good for rarely used widgets).

6.If a dynamic import fails, it throws an error. Use an Error Boundary to catch and display a fallback UI.

7.Webpack assigns chunk names automatically (hash/IDs) unless you specify /\* webpackChunkName: "name" \*/.

8.By default, chunks get numeric or hashed names (0.js, 1.js).

9.You can lazy-load multiple components for the same route using Promise.all or multiple React.lazy() calls.

10.Yes, nested routes can use React.lazy() individually inside <Routes>.

**Webpack Bundle Analyzer**

11. It visualizes bundle size and helps optimize large React apps.

12. Install with npm install --save-dev webpack-bundle-analyzer, then configure in webpack.config.js with new BundleAnalyzerPlugin().

13. It shows file sizes, bundle composition, duplicate code, and optimization opportunities.

14. Open the treemap and check the largest blocks to identify heavy dependencies.

15. Strategies include code splitting, tree shaking, replacing heavy libraries, and dynamic imports.

16. In dev mode, run after webpack --mode development; in production, run after webpack --mode production for accurate sizes.

17. Static mode generates an HTML report, server mode runs an interactive local server with visualization.

18. Use excludeAssets or configure BundleAnalyzerPlugin options to skip certain packages.

19. Tree shaking removes unused exports from dependencies, shrinking bundle size.

20. splitChunks separates vendor and app code, making bundles smaller and analysis easier.

**State Lifting (10)**  
21. Lifting state up means moving state to a common parent so multiple children can share it.

22. Two siblings need lifted state when they must share or sync data (e.g., search input + results list).

23. Pass a callback from parent to child; child calls it to send data up.

24. Drawback: too much state in parent causes complexity and unnecessary re-renders.

25. Lifting reduces prop drilling by keeping shared state closer to where it’s needed.

26. Example: a form with multiple inputs. Parent stores state and passes values + onChange to each input.  
  
function ParentForm() {  
 const [formData, setFormData] = useState({ name: "", email: "" });  
 const handleChange = (e) => setFormData({ ...formData, [e.target.name]: e.target.value });  
 return (<><NameInput value={formData.name} onChange={handleChange} /><EmailInput value={formData.email} onChange={handleChange} /></>);  
}

27. Use techniques like React.memo for child components, useCallback for passing stable functions, and useMemo for derived values so only the components that depend on the changed state re-render.

28. Lift the state to a common parent, then wrap children in a Context.Provider so they can access the shared state and updater function without prop drilling.

29. Because each render creates new function references. useCallback memoizes the function so children receiving it as a prop don’t re-render unnecessarily.

30. By keeping the parent as the single source of truth, passing both value and onChange down to inputs. This preserves controlled behavior while still centralizing the state.