Assignment 11

Q: What is prop drilling in React?

A:
Prop drilling refers to the process of passing data from a parent component down through multiple levels of nested child components, even when intermediate components don't need that data themselves. This occurs because React enforces a unidirectional data flow — data is passed from parent to child using props. In complex applications with deeply nested components, this can lead to unnecessary prop forwarding, making the code harder to maintain and increasing the likelihood of bugs. Prop drilling is often addressed using solutions like the Context API or state management libraries (e.g., Redux) to share data more efficiently.

Q: What is lifting the state up in React?

A:
Lifting state up refers to the process of moving a shared state to the closest common ancestor of two or more components that need to access or modify it. Since React follows unidirectional data flow (from parent to child), when a child component needs to update state that affects its parent or sibling components, the state is "lifted" to the parent component. The parent then passes the state and its updater function (usually from useState) down to the child components via props. This allows the child to trigger updates that are reflected across the relevant parts of the component tree.

Q: What is a Context Provider and Context Consumer in React?

A:
In React, the Context API allows you to share data (like theme, user info, or app settings) across the component tree without manually passing props down at every level.

- **Context Provider**: A Context Provider is a component that makes a specific context value available to all its descendant components. It wraps a part of the component tree and takes a value prop, which is the data to be shared.
- **Context Consumer**: A Context Consumer is used to access the context data inside a component. It reads the current value from the nearest matching Provider above it in the tree.

```jsx
const ThemeContext = React.createContext(); // Step 1: Create Context

function App() {
  return (
    <ThemeContext.Provider value="dark"> {/* Step 2: Provide Context */}
      <ChildComponent />
    </ThemeContext.Provider>
  );
}

function ChildComponent() {
  return (
    <ThemeContext.Consumer>
      {value => <div>Theme is {value}</div>} {/* Step 3: Consume Context */}
    </ThemeContext.Consumer>
  );
}
```

Alternatively, in functional components, you can use the `useContext` hook for a cleaner syntax:

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

Q: Can we use useContext instead of a Context Consumer in functional components?

A:

Yes, in functional components, it's recommended to use the useContext hook instead of the <Context.Consumer> component. The useContext hook provides a simpler and more readable way to access context values.

```jsx
const MyContext = React.createContext();

function MyComponent() {
  const value = useContext(MyContext); // Access the context directly
  return <div>{value}</div>;
}
```

This approach avoids the need for nested render functions (used with `<Context.Consumer>`) and results in cleaner, more maintainable code. However, it can only be used inside functional components or custom hooks.

Q: If you don’t pass a value to a Context Provider, does it take the default value?

A:

Yes, if a component consumes a context and there is **no matching Provider above it in the component tree**, then it will use the **default value** defined when the context was created using React.createContext(defaultValue).

However, if a Provider is used but no value is explicitly passed, the context value becomes undefined, because you’ve technically provided a value of undefined.

```
const MyContext = React.createContext("default");

function App() {
  return (
    // No Provider at all
    <ChildComponent />
  );
}

function ChildComponent() {
  const value = useContext(MyContext);
  return <div>{value}</div>; // Output: "default"
}
```

In this case, value inside ChildComponent would be undefined, because the provider exists but wasn’t given a value.

```
function App() {
  return (
    <MyContext.Provider> {/* No value prop! */}
      <ChildComponent />
    </MyContext.Provider>
  );
}
```