✅ 1. Memoized Fibonacci (useMemo)

🔍 Problem

We want to calculate the nth Fibonacci number. Since Fibonacci is a computationally expensive recursive operation, we want to avoid recalculating it on every render.

💡 Concept: `useMemo`

`useMemo` caches the result of a computation and only re-evaluates it when its dependencies change.

🧠 Approach

- Use `useMemo` to wrap the Fibonacci calculation.
- Only recompute if the input `n` changes.

✅ Code

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

function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

export default function FibonacciCalculator() {
  const [num, setNum] = useState(0);

  const result = useMemo(() ⇒ fibonacci(num), [num]);

  return (
    <div>
    <inputtype="number"
      value={num}
      onChange={e => setNum(+e.target.value)}
    />
    <p>Fibonacci({num}) = {result}</p>
  </div>
 );
}
```

✅ 2. Filter Users List (useMemo)

🔍 Problem

Filter a large array (10,000+ names) based on user search input without re-computing on every render.

💡 Concept: `useMemo`

This avoids re-filtering the entire list unless the search term changes.

🧠 Approach

- Generate the users once (outside component or in `useMemo` without dependencies).
- Use `useMemo` for filtered results depending on `searchTerm`.

✅ Code

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

const users = Array.from({ length: 10000 }, (_, i) => `User ${i}`);

export default function UserSearch() {
  const [searchTerm, setSearchTerm] = useState('');

  const filtered = useMemo(() => {
    return users.filter(name =>
      name.toLowerCase().includes(searchTerm.toLowerCase())
    );
  }, [searchTerm]);

  return (
    <div>
      <input value={searchTerm} onChange={e => setSearchTerm(e.target.valu
e)} />
      <ul>
        {filtered.slice(0, 20).map((name, idx) => (
          <li key={idx}>{name}</li>
        ))}
      </ul>
    </div>
  );
}
```

✅ 3. Debounced Search (useRef)

🔍 Problem

Call an API only after user stops typing for 500ms.

💡 Concept: `useRef`

Store the timeout ID across renders without triggering re-renders.

🧠 Approach

- Store `setTimeout`'s ID in a ref.
- Reset timer on every keypress.

✅ Code

```
jsx
CopyEdit
import React, { useRef, useState } from 'react';

export default function DebouncedSearch() {
  const [term, setTerm] = useState('');
  const timer = useRef(null);

  const handleChange = (e) => {
    const value = e.target.value;
    setTerm(value);

    clearTimeout(timer.current);
    timer.current = setTimeout(() => {
      console.log('Searching for:', value);
    }, 500);
  };

  return <input value={term} onChange={handleChange} />;
}
```

✅ 4. Save Input without Re-render (useRef)

🔍 Problem

Typing in an input should not re-render the component until a submit button is clicked.

💡 Concept: `useRef`

Avoid state updates during typing by using a ref to hold the value.

🧠 Approach

- Use `inputRef.current.value` instead of `useState`.
- Only trigger re-render on submit.

✅ Code

```
jsx
CopyEdit
import React, { useRef, useState } from 'react';

export default function UncontrolledForm() {
  const inputRef = useRef();
  const [submitted, setSubmitted] = useState('');

  const handleSubmit = () => {
    setSubmitted(inputRef.current.value);
  };

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={handleSubmit}>Submit</button>
      <p>Submitted Value: {submitted}</p>
    </div>
  );
}
```

✅ 5. Stopwatch with Ref (useRef)

🔍 Problem

Build a simple stopwatch using `setInterval`. Avoid stale closures.

💡 Concept: `useRef`

Use it to store the interval ID so it survives re-renders without causing them.

🧠 Approach

- Start interval with `intervalRef.current = setInterval(...)`
- Use `clearInterval` to pause/reset

✅ Code

```
jsx
CopyEdit
import React, { useState, useRef } from 'react';

export default function Stopwatch() {
  const [time, setTime] = useState(0);
  const intervalRef = useRef(null);

  const start = () => {
    if (intervalRef.current !== null) return;
    intervalRef.current = setInterval(() => setTime(t => t + 1), 1000);
  };

  const pause = () => {
    clearInterval(intervalRef.current);
    intervalRef.current = null;
  };

  const reset = () => {
    pause();
    setTime(0);
  };

  return (
    <div>
      <h2>{time}s</h2>
      <button onClick={start}>Start</button>
      <button onClick={pause}>Pause</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
```

✅ 6. Callback Optimizer (useCallback)

🔍 Problem

When passing a function as a prop to a memoized child (`React.memo()`), a new function reference on every render causes unnecessary re-renders.

💡 Concept: `useCallback`

`useCallback(fn, deps)` returns a memoized version of the callback that only changes if dependencies change.

🧠 Approach

- Wrap the function using `useCallback`.
- Use `React.memo` for child to observe optimizations.

✅ Code

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

const Child = React.memo(({ onClick }) ⇒ {
  console.log("Child Rendered");
  return <button onClick={onClick}>Click Me</button>;
});

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

  const handleClick = useCallback(() ⇒ {
    console.log("Clicked");
  }, []); // No dependencies

  return (
    <div>
      <p>Parent Count: {count}</p>
      <button onClick={() => setCount(c => c + 1)}>Increment</button>
      <Child onClick={handleClick} />
    </div>
  );
}
```

✅ 7. Memoized Toggle Handler (useCallback)

🔍 Problem

Create a toggle button without recreating the toggle function on every render.

💡 Concept: `useCallback`

Keeps a stable function reference across renders.

🧠 Approach

- Use `useCallback` to wrap the toggle function.
- Prevents unnecessary updates in child components receiving this function.

✅ Code

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

function ToggleButton({ onToggle }) {
  console.log("ToggleButton Rendered");
  return <button onClick={onToggle}>Toggle</button>;
}

const MemoToggleButton = React.memo(ToggleButton);

export default function ToggleParent() {
  const [on, setOn] = useState(false);
  const toggle = useCallback(() ⇒ setOn(o ⇒ !o), []);

  return (
    <div>
      <p>{on ? "ON" : "OFF"}</p>
      <MemoToggleButton onToggle={toggle} />
    </div>
  );
}
```

✅ 8. Lazy Load Dashboard (React.lazy)

🔍 Problem

Only load the Dashboard component when the user navigates to `/dashboard`.

💡 Concept: `React.lazy` + `Suspense`

- Code-splitting for performance.
- `React.lazy` defers loading until needed.
- `Suspense` shows fallback during loading.

🧠 Approach

- Use `React.lazy()` to import Dashboard.
- Wrap it in `<Suspense fallback={...}>`.

✅ Code

```
jsx
CopyEdit
import React, { Suspense } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Dashboard = React.lazy(() ⇒ import('./Dashboard'));

export default function App() {
  return (
    <Router>
      <Suspense fallback={<p>Loading...</p>}>
        <Routes>
          <Route path="/dashboard" element={<Dashboard />} />
        </Routes>
      </Suspense>
    </Router>
  );
}
```

✅ 9. Lazy Import Image Gallery (React.lazy + Suspense)

🔍 Problem

You want to lazy-load individual image components to reduce initial bundle size.

💡 Concept: Lazy-load components individually.

🧠 Approach

- Create `<LazyImage />` components using `React.lazy`.
- Wrap each in `Suspense`.

✅ Code

```jsx
jsx
CopyEdit
import React, { Suspense } from 'react';

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

export default function Gallery() {
  return (
    <div>
      <Suspense fallback={<p>Loading Image...</p>}>
        <LazyImage src="img1.jpg" />
        <LazyImage src="img2.jpg" />
      </Suspense>
    </div>
  );
}
```

✅ 10. Theme Context (Context API)

🔍 Problem

Implement global theme switching.

💡 Concept: `Context API`

Allows sharing state across many components without prop drilling.

🧠 Approach

- Create `ThemeContext` with a toggle function.
- Wrap root component with `ThemeProvider`.

✅ Code

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

const ThemeContext = createContext();

export function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");
  const toggle = () => setTheme(t => (t === "light" ? "dark" : "light"));

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

export function useTheme() {
  return useContext(ThemeContext);
}
```

✅ 11. Auth Context Setup (Context API)

🔍 Problem

Manage login/logout globally.

💡 Concept: Context to share `auth` state and logic.

🧠 Approach

- Store `isAuthenticated`, `login`, `logout` in context.
- Wrap app with `AuthProvider`.

✅ Code

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

const AuthContext = createContext();

export function AuthProvider({ children }) {
  const [isAuthenticated, setAuth] = useState(false);

  const login = () ⇒ setAuth(true);
  const logout = () ⇒ setAuth(false);

  return (
    <AuthContext.Provider value={{ isAuthenticated, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

export function useAuth() {
  return useContext(AuthContext);
}
```

✅ 12. Ref Focus Handler (useRef)

🔍 Problem

Focus an input field when a button is clicked.

💡 Concept: `useRef`

Access the DOM directly.

✅ Code

```
jsx
CopyEdit
import React, { useRef } from 'react';

export default function FocusInput() {
  const inputRef = useRef();

  const focus = () ⇒ inputRef.current.focus();

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={focus}>Focus</button>
    </div>
  );
}
```

✅ 13. Dynamic Form with Ref (useRef)

🔍 Problem

Manage multiple input fields dynamically.

💡 Concept: `useRef([])`

Use array of refs for scalable access.

✅ Code

```
jsx
CopyEdit
import React, { useRef } from 'react';
```

```jsx
export default function DynamicForm() {
  const inputRefs = useRef([]);

  const handleSubmit = () ⇒ {
    inputRefs.current.forEach((ref, i) ⇒ {
      console.log(`Input ${i}:`, ref.value);
    });
  };

  return (
    <div>
      {[...Array(3)].map((_, i) ⇒ (
        <inputkey={i}
          ref={el ⇒ (inputRefs.current[i] = el)}
          placeholder={`Input ${i + 1}`}
        />
      ))}
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
}
```

✅ 14. Memoized Complex Sort (useMemo)

🔍 Problem

Avoid re-sorting a list unless `sortBy` changes.

💡 Concept: `useMemo`

✅ Code

```
jsx
CopyEdit
```

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

const data = [
  { name: "Zoe", age: 25 },
  { name: "Alice", age: 30 },
  { name: "Bob", age: 20 }
];

export default function SortableList() {
  const [sortBy, setSortBy] = useState("name");

  const sortedData = useMemo(() => {
    console.log("Sorting...");
    return [...data].sort((a, b) => a[sortBy].localeCompare?.(b[sortBy]) ?? a[sort
By] - b[sortBy]);
  }, [sortBy]);

  return (
    <div>
      <button onClick={() => setSortBy("name")}>Sort by Name</button>
      <button onClick={() => setSortBy("age")}>Sort by Age</button>
      <ul>
        {sortedData.map((d, i) => (
          <li key={i}>{d.name} - {d.age}</li>
        ))}
      </ul>
    </div>
  );
}
```

✅ 15. Toggle Sidebar with Context (Context API)

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

const SidebarContext = createContext();

export function SidebarProvider({ children }) {
  const [isOpen, setIsOpen] = useState(false);
  const toggle = () => setIsOpen(prev => !prev);

  return (
    <SidebarContext.Provider value={{ isOpen, toggle }}>
      {children}
    </SidebarContext.Provider>
  );
}

export function useSidebar() {
  return useContext(SidebarContext);
}
```

✅ 16. Count Renders with Ref (useRef)

```jsx
jsx
CopyEdit
import React, { useRef } from 'react';

export default function RenderCounter() {
  const renderCount = useRef(1);
  renderCount.current++;

  return <div>Rendered {renderCount.current} times</div>;
}
```

✅ 17. Lazy Load Routes (React.lazy + React Router)

Same as #8 but for multiple routes:

```
jsx
CopyEdit
const Home = React.lazy(() ⇒ import('./Home'));
const About = React.lazy(() ⇒ import('./About'));

<Suspense fallback={<div>Loading...</div>}>
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
  </Routes>
</Suspense>
```

✅ 18. Avoid Inline Function (useCallback)

```
jsx
CopyEdit
const handleDelete = useCallback(() ⇒ {
  console.log("Deleted");
}, [id]);
```

This prevents re-creating the function every render.

✅ 19. Custom Hook with useRef — usePrevious

```
jsx
CopyEdit
import { useEffect, useRef } from 'react';

export function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
    ref.current = value;
  });
  return ref.current;
}
```

Usage:

```
jsx
CopyEdit
const prevCount = usePrevious(count);
```

✅ 20. Context with Reducer (Context API + useReducer)

```
jsx
CopyEdit
const CounterContext = createContext();

const reducer = (state, action) => {
  switch (action.type) {
    case "INC": return { count: state.count + 1 };
    case "DEC": return { count: state.count - 1 };
    case "RESET": return { count: 0 };
    default: return state;
  }
};

export function CounterProvider({ children }) {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <CounterContext.Provider value={{ state, dispatch }}>
      {children}
    </CounterContext.Provider>
  );
}
```