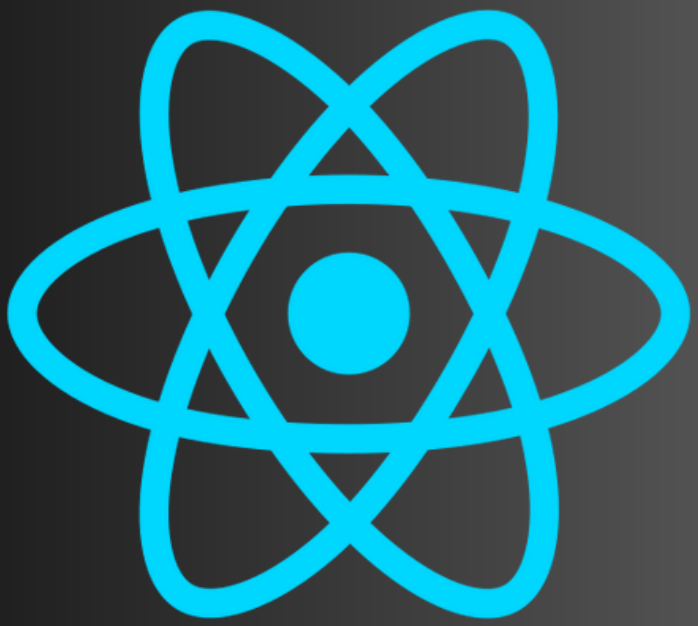

20 React Coding Interview Questions

Complete Solutions with Code Examples & Explanations

Question 1

Easy

Create a counter component with increment, decrement, and reset functionality.

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

functionModal({isOpen,onClose, children }) {
  useEffect(() => {
    consthandleEscape  =(e) => {
      if(e.key==='Escape') onClose();
    };

    if (isOpen) {
      document.addEventListener('keydown', handleEscape);
      document.body.style.overflow = 'hidden';
    }

    return () => {
      document.removeEventListener('keydown', handleEscape);
      document.body.style.overflow = 'unset';
    };
  }, [isOpen, onClose]);

  if (!isOpen) return null;

  return ReactDOM.createPortal(
    <div style={{
      position: 'fixed',
      top: 0,
      left: 0,
      right: 0,
      bottom: 0,
      backgroundColor: 'rgba(0,0,0,0.5)',
      display: 'flex',
      alignItems: 'center',
      justifyContent: 'center'
    }} onClick={onClose}>
      <div style={{
        backgroundColor: 'white',
        padding: '20px',
        borderRadius: '8px',
        maxWidth: '500px'
      }} onClick={(e) => e.stopPropagation()}>
        {children}
        <button onClick={onClose}>Close</button>
      </div>
```

```
    </div>,
    document.body
  );
}

// Usage
functionApp() {
  const[isModalOpen, setIsModalOpen] = useState(false);

  return (
    <div>
      <button onClick={() => setIsModalOpen(true)}>
        Open Modal
      </button>

      <Modal isOpen={isModalOpen} onClose={() => setIsModalOpen(false)}>
        <h2>Modal Title</h2>
        <p>This is modal content</p>
      </Modal>
    </div>
  );
}
```

Explanation

Key Concepts:

- **Portals:** Renders modal outside parent DOM hierarchy using ReactDOM.createPortal.
- **Event Listeners:** Escape key to close modal.
- **stopPropagation:** Prevents closing when clicking modal content.
- **Body Overflow:** Prevents background scrolling when modal is open.
- **Cleanup:** Removes event listeners and restores scroll on unmount.

Accessibility: Should add focus trap and ARIA attributes in production.

Portals | Modal | Event Handling

Question 2

Medium

Build a Todo List with add and delete functionality.

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

function TodoList() {
  const [todos, setTodos] = useState([]);
  const [input, setInput] = useState('');

  const addTodo = () => {
    if (input.trim()) {
      setTodos([...todos,{id:Date.now(), text: input }]);
      setInput('');
    }
  };

  const deleteTodo = (id) => {
    setTodos(todos.filter(todo =>todo.id !== id));
  };

  return (
    <div>
      <input
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Enter todo"
      />
      <button onClick={addTodo}>Add</button>

      <ul>
        {todos.map(todo => (
          <li key={todo.id}>
            {todo.text}
            <button onClick={() => deleteTodo(todo.id)}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default TodoList;
```

Explanation

Key Concepts:

- **Array State:** Managing array of objects in state.
- **Controlled Input:** Input value controlled by React state.
- **Key Prop:** Using unique id for list items (essential for React's reconciliation).
- **Immutability:** Using spread operator and filter to create new arrays.

Best Practice: Always use unique, stable IDs for keys, not array index.

Lists | Array State | Controlled Components

Question 3

Medium

Create a custom hook for fetching data from an API.

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

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

  useEffect(() => {
    constfetchData = async () => {
      try{
        setLoading(true);
        const response = await fetch(url);
        if(!response.ok) throw new Error('Network response was not ok');
        const json = await response.json();
        setData(json);
        setError(null);
      }catch (err) {
        setError(err.message);
        setData(null);
      }finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return{data, loading, error };
}

// Usage
functionUserList() {
  const{data, loading, error } = useFetch('https://api.example.com/users');

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

  return (
    <ul>
      {data.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}
```

Explanation

Key Concepts:

- **Custom Hooks:** Reusable logic extraction. Must start with "use".
- **useEffect:** Runs side effects (API calls) after render.
- **Async/Await:** Handling promises for cleaner async code.
- **Error Handling:** Try-catch for API errors.
- **Loading States:** Managing loading, data, and error states.

Improvement: Add cleanup with AbortController to cancel requests on unmount.

Custom Hooks | useEffect | API Calls | Async

Question 4

Easy

Implement a toggle button component that shows/hides content.

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

functionToggle() {
  const[isVisible, setIsVisible] = useState(false);

  consttoggleVisibility = () => setIsVisible(!isVisible);

  return (
    <div>
      <button onClick={toggleVisibility}>
        {isVisible ? 'Hide' : 'Show'} Content
      </button>

      {isVisible && (
        <div>
          <p>This is the toggled content!</p>
        </div>
      )}
    </div>
  );
}

export default Toggle;
```

💡 Explanation

Key Concepts:

- **Boolean State:** Using boolean for toggle functionality.
- **Conditional Rendering:** Using && operator for showing/hiding content.
- **Dynamic Button Text:** Ternary operator for conditional text.

Alternative: Could also use ternary with null: {isVisible ? <Content /> : null}

Conditional Rendering | Boolean State | Toggle

Question 5

Medium

Create a form with validation (name, email, password).

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

functionForm() {
  const[formData, setFormData] = useState({
    name: '',
    email: '',
    password: ''
  });
  const[errors, setErrors] = useState({});

  constvalidate = () => {
    constnewErrors = {};

    if(!formData.name.trim()) {
      newErrors.name = 'Name is required';
    }

    if(!formData.email.includes('@')) {
      newErrors.email = 'Valid email is required';
    }

    if(formData.password.length < 6) {
      newErrors.password = 'Password must be at least 6 characters';
    }

    return newErrors;
  };

  consthandleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value
    });
  };

  consthandleSubmit = (e) => {
    e.preventDefault();
    constvalidationErrors = validate();

    if(Object.keys(validationErrors).length === 0) {
      console.log('Form submitted:', formData);
      //Submit to API
    } else {
      setErrors(validationErrors);
    }
  };
```

```
  return (
    <form onSubmit={handleSubmit}>
      <div>
        <input
          name="name"
          value={formData.name}
          onChange={handleChange}
          placeholder="Name"
        />
        {errors.name && <span style={{color: 'red'}}>{errors.name}</span>}
      </div>

      <div>
        <input
          name="email"
          value={formData.email}
          onChange={handleChange}
          placeholder="Email"
        />
        {errors.email && <span style={{color: 'red'}}>{errors.email}</span>}
      </div>

      <div>
        <input
          type="password"
          name="password"
          value={formData.password}
          onChange={handleChange}
          placeholder="Password"
        />
        {errors.password && <span style={{color: 'red'}}>{errors.password}</span>}
      </div>

      <button type="submit">Submit</button>
    </form>
  );
}

export default Form;
```

💡 Explanation

Key Concepts:

- **Object State:** Managing multiple form fields in single state object.
- **Computed Property Names:** [e.target.name] for dynamic object keys.
- **Form Validation:** Client-side validation before submission.
- **preventDefault:** Stops default form submission behavior.
- **Error Display:** Conditional rendering of error messages.

Question 6

Medium

Build a search filter component for a list of items.

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

functionSearchFilter() {
  const[searchTerm, setSearchTerm] = useState('');

  const items = [
    {id:1,name: 'Apple', category: 'Fruit' },
    {id:2,name: 'Carrot', category: 'Vegetable' },
    {id:3,name: 'Banana', category: 'Fruit' },
    {id:4,name: 'Broccoli', category: 'Vegetable' }
  ];

  constfilteredItems = items.filter(item =>
    item.name.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      <input
        type="text"
        placeholder="Search items..."
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
      />

      <ul>
        {filteredItems.map(item => (
          <li key={item.id}>
            {item.name} - {item.category}
          </li>
        ))}
      </ul>

      {filteredItems.length === 0 && (
        <p>Noitems found</p>
      )}
    </div>
  );
}

export default SearchFilter;
```

Explanation

Key Concepts:

- **Filter Method:** Creating new filtered array based on search term.
- **Case-Insensitive Search:** Converting to lowercase for comparison.
- **Controlled Input:** Search input controlled by state.
- **Dynamic Rendering:** List updates automatically as user types.

Performance Tip: For large lists, consider debouncing search input or using useMemo.

Filtering Search Array Methods

Question 7

Hard

Implement a debounced search input using custom hook.

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

functionuseDebounce(value, delay) {
  const[debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    consthandler=setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}

functionDebouncedSearch() {
  const[searchTerm, setSearchTerm] = useState('');
  constdebouncedSearchTerm = useDebounce(searchTerm, 500);

  useEffect(() => {
    if(debouncedSearchTerm) {
      //Make API call with debounced value
      console.log('Searching for:', debouncedSearchTerm);
      // fetch(`/api/search?q=${debouncedSearchTerm}`)
    }
  }, [debouncedSearchTerm]);

  return (
    <div>
      <input
        type="text"
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
        placeholder="Search..."
      />
      <p>Searching for: {debouncedSearchTerm}</p>
    </div>
  );
}

export default DebouncedSearch;
```

Explanation

Key Concepts:

- **Debouncing:** Delays execution until user stops typing.
- **setTimeout:** Delays state update by specified time.
- **Cleanup Function:** clearTimeout prevents memory leaks and cancels previous timers.
- **Custom Hook Pattern:** Reusable debounce logic.

Use Case: Reduces API calls - only searches after 500ms of no typing.

Debouncing | Custom Hooks | Performance

Question 8

Medium

Create a modal component that can be opened and closed.

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

functionModal({isOpen,onClose, children }) {
  useEffect(() => {
    consthandleEscape  =(e) => {
      if(e.key==='Escape') onClose();
    };

    if (isOpen) {
      document.addEventListener('keydown', handleEscape);
      document.body.style.overflow = 'hidden';
    }

    return () => {
      document.removeEventListener('keydown', handleEscape);
      document.body.style.overflow = 'unset';
    };
  }, [isOpen, onClose]);

  if (!isOpen) return null;

  return ReactDOM.createPortal(
    <div style={{
      position: 'fixed',
      top: 0,
      left: 0,
      right: 0,
      bottom: 0,
      backgroundColor: 'rgba(0,0,0,0.5)',
      display: 'flex',
      alignItems: 'center',
      justifyContent: 'center'
    }} onClick={onClose}>
      <div style={{
        backgroundColor: 'white',
        padding: '20px',
        borderRadius: '8px',
        maxWidth: '500px'
      }} onClick={(e) => e.stopPropagation()}>
        {children}
        <button onClick={onClose}>Close</button>
      </div>
    </div>,
    document.body
  );
}
```

```
// Usage
functionApp() {
  const[isModalOpen, setIsModalOpen] = useState(false);

  return (
    <div>
      <button onClick={() => setIsModalOpen(true)}>
        Open Modal
      </button>

      <Modal isOpen={isModalOpen} onClose={() => setIsModalOpen(false)}>
        <h2>Modal Title</h2>
        <p>This is modal content</p>
      </Modal>
    </div>
  );
}
```

💡 Explanation

Key Concepts:

Question 9

Medium

Build a pagination component for a list of items.

function Counter() { const [count, setCount] = useState(0); const increment = () => setCount(count + 1); const decrement = () => setCount(count - 1); const reset = () => setCount(0); return (<h2>Count: {count}</h2> <button onClick={increment}>Increment</button> <button onClick={decrement}>Decrement</button> <button onClick={reset}>Reset</button>); } export default Counter;

💡 Explanation

Key Concepts:

- **Portals:** Renders modal outside parent DOM hierarchy using ReactDOM.createPortal.
- **Event Listeners:** Escape key to close modal.
- **stopPropagation:** Prevents closing when clicking modal content.
- **Body Overflow:** Prevents background scrolling when modal is open.
- **Cleanup:** Removes event listeners and restores scroll on unmount.

Accessibility: Should add focus trap and ARIA attributes in production.

Portals | Modal | Event Handling

Question 8

Medium

Create a modal component that can be opened and closed.

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

function Pagination() {
  const[currentPage,setCurrentPage] = useState(1);
  const itemsPerPage = 5;

  constitems=Array.from({ length: 50 }, (_, i) => ({
    id: i + 1,
    name: `Item ${i + 1}`
  }));

  consttotalPages  =Math.ceil(items.length / itemsPerPage);
  conststartIndex  =(currentPage - 1) * itemsPerPage;
  constendIndex=   startIndex + itemsPerPage;
  constcurrentItems=   items.slice(startIndex, endIndex);

  constgoToPage=   (page)=> {
    setCurrentPage(Math.max(1, Math.min(page, totalPages)));
  };

  return (
    <div>
      <ul>
        {currentItems.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>

      <div>
        <button
          onClick={()=>goToPage(currentPage - 1)}
          disabled={currentPage === 1}
        >
          Previous
        </button>

        <span> Page{currentPage} of {totalPages} </span>

        <button
          onClick={()=>goToPage(currentPage + 1)}
          disabled={currentPage === totalPages}
        >
          Next
        </button>
      </div>

      <div>
```

```
        {Array.from({ length: totalPages }, (_, i) => (
          <button
            key={i + 1}
            onClick={() => goToPage(i + 1)}
            style={{
              fontWeight: currentPage === i + 1 ? 'bold' : 'normal'
            }}
          >
            {i + 1}
          </button>
        ))}
      </div>
    </div>
  );
}

export default Pagination;
```

💡 Explanation

Key Concepts:

- **Slice Method:** Extracting subset of items for current page.
- **Math.ceil:** Calculating total pages (rounds up).
- **Boundary Checks:** Math.max/min ensures page stays within valid range.
- **Disabled State:** Disables buttons at boundaries.
- **Dynamic Array:** Array.from creates page number buttons.

Enhancement: Could add "jump to page" input or show limited page numbers (1...5...10).

Pagination | Array Methods | Navigation

Question 10

Hard

Implement an infinite scroll component.

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

functionInfiniteScroll() {
  const[items, setItems] = useState([]);
  const[page, setPage] = useState(1);
  const[loading, setLoading] = useState(false);
  const[hasMore, setHasMore] = useState(true);
  constobserver = useRef();

  constlastItemRef = useCallback(node => {
    if(loading) return;
    if(observer.current) observer.current.disconnect();

    observer.current = new IntersectionObserver(entries => {
      if(entries[0].isIntersecting && hasMore) {
        setPage(prevPage => prevPage + 1);
      }
    });

    if(node) observer.current.observe(node);
  },[loading, hasMore]);

  useEffect(() => {
    constfetchItems = async () => {
      setLoading(true);

      //Simulate API call
      setTimeout(() => {
        const newItems = Array.from({ length: 20 }, (_, i) => ({
          id: (page - 1) * 20 + i + 1,
          name: `Item ${(page - 1) * 20 + i + 1}`
        }));

        setItems(prev => [...prev, ...newItems]);
        setHasMore(page < 10); // Stop after 10 pages
        setLoading(false);
      }, 1000);
    };

    fetchItems();
  }, [page]);

  return (
    <div>
      {items.map((item, index) => {
        if(items.length === index + 1) {
          return (
```

```
            <divref={lastItemRef} key={item.id}>
              {item.name}
            </div>
          );
        }
        return <div key={item.id}>{item.name}</div>;
      })}

      {loading && <div>Loading...</div>}
      {!hasMore&&  <div>No more items</div>}
    </div>
  );
}

export default InfiniteScroll;
```

💡 Explanation

Key Concepts:

- **IntersectionObserver:** Detects when last item enters viewport.
- **useCallback:** Memoizes ref callback to prevent unnecessary re-creation.
- **useRef:** Stores observer instance across renders.
- **Cleanup:** Disconnects observer before creating new one.
- **Conditional Ref:** Only last item gets the ref for observation.

Performance: Much more efficient than scroll event listeners.

Infinite Scroll | IntersectionObser ver | useCallback

Question 11

Medium

Create a tabs component with multiple tab panels.

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

function Tabs() {
const [activeTab, setActiveTab] = useState(0);

  const tabs = [
    { label: 'Tab 1', content: 'Content for Tab 1' },
    { label: 'Tab 2', content: 'Content for Tab 2' },
    { label: 'Tab 3', content: 'Content for Tab 3' }
 ];

  return (
    <div>
      <divstyle={{display:'flex',borderBottom:     '2px solid #ccc' }}>
        {tabs.map((tab, index) => (
          <button
            key={index}
            onClick={() => setActiveTab(index)}
            style={{
              padding: '10px 20px',
              border: 'none',
              background:activeTab===index    ?'#007bff' : '#f0f0f0',
              color: activeTab === index ? 'white' : 'black',
              cursor: 'pointer',
              borderRadius: '4px 4px 0 0'
            }}
          >
            {tab.label}
          </button>
        ))}
      </div>

      <divstyle={{padding:'20px',border:'1pxsolid #ccc' }}>
        {tabs[activeTab].content}
      </div>
    </div>
  );
}

export default Tabs;
```

Explanation

Key Concepts:

- **Index-based State:** Using numeric index to track active tab.
- **Dynamic Styling:** Conditional styles based on active state.
- **Array Indexing:** Accessing content using activeTab index.
- **Map Method:** Dynamically rendering tab buttons from array.

Advanced Version: Could create compound components (Tabs, TabList, Tab, TabPanel) for more flexibility.

Tabs Navigation Conditional Styling

Question 12

Hard

Build a drag and drop list reordering component.

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

functionDragDropList() {
  const[items, setItems] = useState([
    {id:1,  text:  'Item  1'  },
    {id:2,  text:  'Item  2'  },
    {id:3, text: 'Item 3' },
    {id:4, text: 'Item 4' }
  ]);
  const[draggedItem, setDraggedItem] = useState(null);

  consthandleDragStart = (e, item) => {
    setDraggedItem(item);
    e.dataTransfer.effectAllowed = 'move';
  };

  consthandleDragOver = (e) => {
    e.preventDefault();
    e.dataTransfer.dropEffect = 'move';
  };

  consthandleDrop = (e, targetItem) => {
    e.preventDefault();

    if(!draggedItem || draggedItem.id === targetItem.id) return;

    constdraggedIndex = items.findIndex(item => item.id === draggedItem.id);
    consttargetIndex = items.findIndex(item => item.id === targetItem.id);

    constnewItems = [...items];
    newItems.splice(draggedIndex, 1);
    newItems.splice(targetIndex, 0, draggedItem);

    setItems(newItems);
    setDraggedItem(null);
  };

  return (
    <div>
      {items.map(item => (
        <div
          key={item.id}
          draggable
          onDragStart={(e) => handleDragStart(e, item)}
          onDragOver={handleDragOver}
          onDrop={(e) => handleDrop(e, item)}
          style={{
```

```
            padding: '15px',
            margin: '5px',
            backgroundColor: draggedItem?.id === item.id ? '#e0e0e0' : '#f5f5f5',
            border: '1px solid #ddd',
            cursor: 'move',
            userSelect: 'none'
          }}
        >
          {item.text}
        </div>
      ))}
    </div>
  );
}

export default DragDropList;
```

💡 Explanation

Key Concepts:

- **HTML5 Drag and Drop API:** Using draggable attribute and drag events.
- **dataTransfer:** Browser API for drag data transfer.
- **preventDefault:** Required on dragOver to allow dropping.
- **Array Manipulation:** splice to remove and insert items at new position.
- **Visual Feedback:** Highlighting dragged item.

Library Alternative: react-beautiful-dnd provides better UX and accessibility.

Drag and Drop | Array Manipulation | HTML5 API

Question 13

Medium

Implement a timer/countdown component.

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

functionTimer() {
  const[seconds, setSeconds] = useState(0);
  const[isActive, setIsActive] = useState(false);
  constintervalRef = useRef(null);

   useEffect(() => {
      if(isActive) {
       intervalRef.current = setInterval(() => {
         setSeconds(s => s + 1);
       }, 1000);
     } else {
       if(intervalRef.current) {
         clearInterval(intervalRef.current);
       }
     }

     return() => {
       if(intervalRef.current) {
         clearInterval(intervalRef.current);
       }
     };
  }, [isActive]);

  consttoggle = () => setIsActive(!isActive);
  constreset = () => {
    setSeconds(0);
    setIsActive(false);
  };

  constformatTime = (totalSeconds) => {
    consthours = Math.floor(totalSeconds / 3600);
    constminutes = Math.floor((totalSeconds % 3600) / 60);
    constsecs = totalSeconds % 60;

    return[hours, minutes, secs]
      .map(val => String(val).padStart(2, '0'))
      .join(':');
  };

  return (
    <div>
      <h2>{formatTime(seconds)}</h2>
      <button onClick={toggle}>
        {isActive ? 'Pause' : 'Start'}
      </button>
```

```
      <button onClick={reset}>Reset</button>
    </div>
  );
}

export default Timer;
```

💡 Explanation

Key Concepts:

- **setInterval:** Executes function repeatedly at specified interval.
- **useRef:** Stores interval ID without causing re-renders.
- **Cleanup:** clearInterval in useEffect return to prevent memory leaks.
- **Functional Update:** setSeconds(s => s + 1) ensures correct value.
- **Time Formatting:** Converting seconds to HH:MM:SS format.

Countdown Version: Start with initial value and decrement instead.

Timer | setInter val | useRef

Question 14

Easy

Create a character counter for a textarea with max limit.

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

functionCharacterCounter() {
  const[text, setText] = useState('');
  constmaxLength = 200;

  consthandleChange = (e) => {
    constvalue = e.target.value;
    if(value.length <= maxLength) {
      setText(value);
    }
  };

  constremaining = maxLength - text.length;
  constisNearLimit = remaining < 20;

  return (
    <div>
      <textarea
        value={text}           onChange=
        {handleChange}
        placeholder="Type something..."
        rows={5}
        style={{
          width: '100%',
          padding: '10px',
          fontSize: '16px'
        }}
      />

      <divstyle={{
        marginTop: '10px',
        color: isNearLimit ? 'red' : 'gray'
      }}>
        {text.length} / {maxLength} characters
        <br />
        {remaining} characters remaining
      </div>
    </div>
  );
}

export default CharacterCounter;
```

Explanation

Key Concepts:

- **Controlled Textarea:** Value controlled by state.
- **Length Validation:** Prevents input beyond max length.
- **Conditional Styling:** Color changes when near limit.
- **Character Counting:** Real-time count display.

Enhancement: Could add word count, show warning at threshold, or allow exceeding with visual indication.

Forms | Validation | Character Counter

Question 15

Medium

Build a star rating component.

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

function StarRating({ totalStars = 5 }) {
const [rating, setRating] = useState(0);
const [hover, setHover] = useState(0);

  return (
    <div style={{ fontSize: '2rem' }}>
      {[...Array(totalStars)].map((_, index) => {
        const starValue = index + 1;

        return (
          <span
            key={index}
            onClick={() => setRating(starValue)}
            onMouseEnter={() => setHover(starValue)}
            onMouseLeave={() => setHover(0)}
            style={{
              cursor: 'pointer',
              color:starValue  <=(hover||   rating) ? '#ffc107' : '#e4e5e9',
              transition: 'color 0.2s'
            }}
          >
            ★
          </span>
        );
      })}

      <div style={{ fontSize: '1rem', marginTop: '10px' }}>
        {rating>0?`You     rated${rating}  out of${totalStars} stars` : 'No rating yet'}
      </div>
    </div>
  );
}

export default StarRating;
```

💡 Explanation

Key Concepts:

- **Array Creation:** [...Array(n)] creates array of n elements.
- **Hover State:** Preview rating before clicking.

- **Conditional Color:** Highlights stars based on rating/hover.
- **Mouse Events:** onMouseEnter/Leave for hover effects.
- **Dynamic Props:** totalStars allows customization.

Enhancement: Add half-star ratings, disable after rating, or show rating distribution.

Rating | Interactive UI | Mouse Events

Question 16

Hard

Implement a custom useLocalStorage hook.

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

functionuseLocalStorage(key,initialValue) {
  //Getinitialvalue    fromlocalStorage or use default
  const[storedValue,setStoredValue] = useState(() => {
    try{
      const item = window.localStorage.getItem(key);
      returnitem?JSON.parse(item) : initialValue;
    } catch (error) {
      console.error(error);
      return initialValue;
    }
  });

  //UpdatelocalStoragewhenvalue changes
  const setValue = (value) => {
    try{
      constvalueToStore=   valueinstanceof Function
        ? value(storedValue)
        : value;

      setStoredValue(valueToStore);
      window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) {
      console.error(error);
    }
  };

  //Syncwith   localStoragechanges from other tabs
  useEffect(() => {
       consthandleStorageChange=(e) => {
      if(e.key===key&&      e.newValue) {
        setStoredValue(JSON.parse(e.newValue));
      }
    };

    window.addEventListener('storage', handleStorageChange);
    return()=>window.removeEventListener('storage', handleStorageChange);
  }, [key]);

  return [storedValue, setValue];
}

// Usage Example
function App() {
  const [name, setName] = useLocalStorage('name', 'Guest');
```

```
  return (
    <div>
      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter your name"
      />
      <p>Hello, {name}!</p>
    </div>
  );
}
```

💡 Explanation

Key Concepts:

- **Lazy Initialization:** Using function in useState to read localStorage once.
- **JSON Serialization:** Storing complex data types as JSON strings.
- **Error Handling:** Try-catch for localStorage quota errors.
- **Storage Event:** Syncs state across browser tabs.
- **Functional Updates:** Supports both direct values and updater functions.

Real-world Use: Persist user preferences, form data, shopping carts, theme settings.

Custom Hooks | localStorage | Persistence

Question 17

Medium

Create an accordion component with expand/collapse functionality.

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

function Accordion() {
  const [openIndex, setOpenIndex] = useState(null);

  const items = [
    {title:'Section1',content:     'Content forsection  1'  },
    {title:'Section2',content:     'Content forsection 2' },
    {title:'Section3',content:     'Content forsection 3' }
  ];

  const toggleItem = (index) => {
    setOpenIndex(openIndex === index ? null : index);
  };

  return (
    <div style={{ width: '400px' }}>
      {items.map((item, index) => (
        <div
          key={index}
          style={{
            border: '1px solid #ddd',
            marginBottom: '5px',
            borderRadius: '4px'
          }}
        >
          <div
            onClick={() => toggleItem(index)}
            style={{
              padding: '15px',
              backgroundColor: '#f5f5f5',
              cursor: 'pointer',
              display: 'flex',
              justifyContent: 'space-between',
              alignItems: 'center',
              fontWeight: 'bold'
            }}
          >
            {item.title}
            <span>{openIndex === index ? '-' : '+'}</span>
          </div>

          {openIndex === index && (
            <divstyle={{padding:   '15px', backgroundColor: 'white' }}>
              {item.content}
            </div>
          )}
```

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

export default Accordion;
```

💡 Explanation

Key Concepts:

- **Single Active Item:** Only one section open at a time.
- **Toggle Logic:** Clicking open item closes it, clicking closed item opens it.
- **Conditional Rendering:** Content only renders when open.
- **Visual Indicator:** +/− symbols show open/closed state.

Variation: Allow multiple sections open by using array of indices instead of single index.

Accordion | Toggle | UI Component

Question 18

Hard

Build a multi-step form wizard with validation.

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

function FormWizard() {
  const[currentStep,setCurrentStep] = useState(0);
  const[formData,  setFormData] = useState({
    name: '',
    email: '',
    address: ''
  });
  const[errors,setErrors] = useState({});

  const steps = [
    {
      label: 'Personal Info',
      fields: ['name'],
      validate:(data)  => {
        const err = {};
        if(!data.name.trim()) err.name = 'Name is required';
        return err;
      }
    },
    {
      label: 'Contact Info',
      fields: ['email'],
      validate:(data)  => {
        const err = {};
        if(!data.email.includes('@')) err.email = 'Valid email required';
        return err;
      }
    },
    {
      label: 'Address',
      fields: ['address'],
      validate:(data)  => {
        const err = {};
        if(!data.address.trim()) err.address = 'Address is required';
        return err;
      }
    }
  ];

  consthandleChange=   (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value
    });
    //Clearerror   for this field
```

```
    setErrors({ ...errors, [e.target.name]: '' });
  };

  const handleNext = () => {
    const stepErrors = steps[currentStep].validate(formData);

    if (Object.keys(stepErrors).length === 0) {
      setCurrentStep(currentStep + 1);
      setErrors({});
    } else {
      setErrors(stepErrors);
    }
  };

  const handlePrevious = () => {
    setCurrentStep(currentStep - 1);
    setErrors({});
  };

  const handleSubmit = () => {
    const stepErrors = steps[currentStep].validate(formData);

    if (Object.keys(stepErrors).length === 0) {
    console.log('Form submitted:', formData);
    alert('Form submitted successfully!');
    } else {
      setErrors(stepErrors);
    }
  };

  const currentStepData = steps[currentStep];
 const isLastStep = currentStep === steps.length - 1;

  return (
    <div style={{ maxWidth: '500px', margin: '0 auto' }}>
      {/* Progress Indicator */}
      <div style={{ display: 'flex', marginBottom: '20px' }}>
        {steps.map((step, index) => (
          <div
            key={index}
            style={{
              flex: 1,
              padding: '10px',
              textAlign: 'center',
              backgroundColor: index===currentStep?    '#007bff' : '#e9ecef',
              color: index === currentStep ? 'white' : 'black',
              fontWeight: index === currentStep ? 'bold' : 'normal'
            }}
          >
            {step.label}
          </div>
        ))}
      </div>

      {/* Form Fields */}
```

```
      <div>
        {currentStepData.fields.map(field => (
          <div key={field} style={{ marginBottom: '15px' }}>
            <label style={{display: 'block', marginBottom: '5px' }}>
              {field.charAt(0).toUpperCase() + field.slice(1)}
            </label>
            <input
              type="text"
              name={field}
              value={formData[field]}
              onChange={handleChange}
              style={{
                width: '100%',
                padding: '8px',
                border:errors[field] ? '2px solid red' : '1px solid #ddd'
              }}
            />
            {errors[field] && (
              <spanstyle={{color: 'red', fontSize: '14px' }}>
                {errors[field]}
              </span>
            )}
          </div>
        ))}
      </div>

      {/* Navigation Buttons */}
      <divstyle={{display:'flex', justifyContent: 'space-between', marginTop: '20px' }}>
        <button
          onClick={handlePrevious}
          disabled={currentStep=== 0}
          style={{
            padding: '10px 20px',
            cursor:currentStep=== 0 ? 'not-allowed' : 'pointer',
            opacity:currentStep=== 0 ? 0.5 : 1
          }}
        >
          Previous
        </button>

        {isLastStep ? (
          <buttononClick={handleSubmit} style={{ padding: '10px 20px' }}>
            Submit
          </button>
        ) : (
          <buttononClick={handleNext} style={{ padding: '10px 20px' }}>
            Next
          </button>
        )}
      </div>
    </div>
  );
}

export default FormWizard;
```

Explanation

Key Concepts:

- **Multi-step Navigation:** Using step index to control which fields display.
- **Step-based Validation:** Each step has its own validation rules.
- **Progress Indicator:** Visual feedback showing current step.
- **State Persistence:** Form data persists across steps.
- **Conditional Buttons:** Shows Submit on last step, Next on others.

Real-world Application: Registration flows, checkout processes, survey forms.

Multi-step Form | Wizard | Validation

Question 19

Medium

Create a theme switcher (light/dark mode) with Context API.

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

// Create Theme Context
const ThemeContext = createContext();

// ThemeProviderComponent
exportfunctionThemeProvider({children }) {
  const [theme, setTheme] = useState('light');

  useEffect(() => {
    // Load theme from localStorage
    constsavedTheme=   localStorage.getItem('theme') || 'light';
    setTheme(savedTheme);
  }, []);

  useEffect(() => {
    // Save theme to localStorage
    localStorage.setItem('theme', theme);
    // Apply theme to document
    document.body.className = theme;
  }, [theme]);

  const toggleTheme = () => {
    setTheme(prevTheme =>prevTheme=== 'light' ? 'dark' : 'light');
  };

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

// Custom hook to use theme
export function useTheme() {
const context = useContext(ThemeContext);
 if (!context) {
    throw new Error('useTheme must be used within ThemeProvider');
  }
 return context;
}

// App Component
function App() {
return (
    <ThemeProvider>
      <ThemedComponent />
```

```
    </ThemeProvider>
  );
}

//Componentthatuses     theme
functionThemedComponent() {
  const{theme,toggleTheme } = useTheme();

  const styles = {
    light: {
      backgroundColor: '#ffffff',
      color: '#000000'
    },
    dark: {
      backgroundColor: '#1a1a1a',
      color: '#ffffff'
    }
  };

  return (
    <div style={{
      ...styles[theme],
      padding: '20px',
      minHeight: '100vh'
    }}>
      <h1>CurrentTheme: {theme}</h1>
      <button onClick={toggleTheme}>
        Switch to{theme === 'light' ? 'Dark' : 'Light'} Mode
      </button>
    </div>
  );
}

export default App;
```

💡 Explanation

Key Concepts:

- **Context API:** Global state management for theme across entire app.
- **Custom Hook:** useTheme provides easy access to theme context.
- **localStorage:** Persists theme preference across sessions.
- **Provider Pattern:** Wraps app to provide theme to all children.
- **Dynamic Styling:** Styles change based on current theme.

Enhancement: Add CSS variables, system preference detection, or multiple theme options.

Question 20

Hard

Implement a shopping cart with add, remove, and quantity update features.

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

// CartReducer
function cartReducer(state, action) {
  switch (action.type) {
    case 'ADD_ITEM':
      constexistingItem=   state.find(item => item.id === action.payload.id);

      if (existingItem) {
        return state.map(item =>
          item.id === action.payload.id
            ?{...item,quantity:    item.quantity + 1 }
            : item
        );
      }

      return[...state,{...action.payload, quantity: 1 }];

    case 'REMOVE_ITEM':
      returnstate.filter(item  =>item.id !== action.payload);

    case 'UPDATE_QUANTITY':
      return state.map(item =>
        item.id === action.payload.id
          ?{...item,   quantity:action.payload.quantity }
          : item
      );

    case 'CLEAR_CART':
      return [];

    default:
      return state;
  }
}

functionShoppingCart() {
  const[cart, dispatch] = useReducer(cartReducer, []);

  constproducts = [
    {id:1, name: 'Laptop', price: 999 },
    {id:2, name: 'Phone', price: 699 },
    {id:3, name: 'Headphones', price: 199 }
  ];
```

```
  const addToCart = (product) => {
    dispatch({ type:'ADD_ITEM',payload: product });
  };

  const removeFromCart = (id) => {
    dispatch({ type:'REMOVE_ITEM',payload: id });
  };

  constupdateQuantity  =(id,quantity) => {
    if (quantity > 0) {
      dispatch({type:  'UPDATE_QUANTITY', payload: { id, quantity } });
    } else {
      removeFromCart(id);
    }
  };

  const clearCart = () => {
    dispatch({ type: 'CLEAR_CART' });
 };

  consttotal=cart.reduce((sum,item) => sum + (item.price * item.quantity), 0);

  return (
    <divstyle={{display:   'flex', gap: '20px', padding: '20px' }}>
      {/* Products */}
      <div style={{ flex: 1 }}>
        <h2>Products</h2>
        {products.map(product => (
          <div key={product.id} style={{
            border: '1px solid #ddd',
            padding: '15px',
            marginBottom: '10px',
            display: 'flex',
            justifyContent: 'space-between',
            alignItems: 'center'
          }}>
            <div>
              <h3>{product.name}</h3>
              <p>${product.price}</p>
            </div>
            <button onClick={() => addToCart(product)}>
              Add to Cart
            </button>
          </div>
        ))}
      </div>

      {/* Cart */}
      <divstyle={{flex:1,border:'1px solid #ddd', padding: '15px' }}>
        <h2>Shopping Cart</h2>

        {cart.length === 0 ? (
          <p>Your cart is empty</p>
        ) : (
          <>
```

```
            {cart.map(item => (
              <div key={item.id} style={{
                borderBottom: '1px solid #eee',
                padding: '10px 0',
                display: 'flex',
                justifyContent: 'space-between',
                alignItems: 'center'
              }}>
                <div>
                  <h4>{item.name}</h4>
                  <p>${item.price} each</p>
                </div>

                <divstyle={{display:   'flex', alignItems: 'center', gap: '10px' }}>
                  <buttononClick={()=>updateQuantity(item.id, item.quantity - 1)}>
                    -
                  </button>
                  <span>{item.quantity}</span>
                  <buttononClick={()=>updateQuantity(item.id, item.quantity + 1)}>
                    +
                  </button>
                  <button onClick={() => removeFromCart(item.id)}>
                    Remove
                  </button>
                </div>
              </div>
            ))}

            <divstyle={{marginTop:'20px', fontSize: '1.2em', fontWeight: 'bold' }}>
              Total: ${total.toFixed(2)}
            </div>

            <button
              onClick={clearCart}
              style={{ marginTop:'10px',width: '100%', padding: '10px' }}
            >
              Clear Cart
            </button>
          </>
        )}
      </div>
    </div>
  );
}

export default ShoppingCart;
```

💡 Explanation

Key Concepts:

- **useReducer:** Complex state logic with multiple actions.

- **Reducer Pattern:** Pure function handling all state updates.
- **Immutable Updates:** Using map/filter for state updates.
- **Quantity Management:** Increment/decrement with validation.
- **Total Calculation:** Using reduce to sum cart items.
- **Duplicate Handling:** Incrementing quantity for existing items.

Production Ready: Add persistence (localStorage), checkout flow, or Context for global cart state.

useReducer Shopping Cart Complex State