1. **What causes unneccessary re renders?**

Unnecessary re-renders in React can be caused by several factors, and they usually happen when a component re-renders even though it doesn't need to. Some common causes include:

1. **State Changes Without Necessity**: React re-renders components whenever their state changes. If we update the state without actually needing to, it triggers a re-render. To avoid this, we should only update the state when it’s truly necessary and when the value has actually changed.
2. **Frequent Prop Changes**: If props passed to a child component change often, it causes that child to re-render every time. One way to avoid this is by using React.memo for functional components or PureComponent for class components. This helps React skip re-renders when the props haven't changed.
3. **Passing New Objects or Arrays as Props**: Objects and arrays are reference types in JavaScript. Every time a new object or array is passed as a prop, React sees it as a change, even if the contents are the same. We can prevent this by using useMemo or useCallback to memoize objects or functions and prevent them from being recreated on each render.
4. **Inline Functions in JSX**: If we pass inline functions or arrow functions directly in JSX, React creates a new function on every render, causing unnecessary re-renders of child components. To fix this, we can use useCallback to memoize the function and avoid passing a new instance each time.
5. **Unnecessary Component Re-renders**: Sometimes components may re-render even though their props or state haven’t changed. We can optimize this by using React.memo for functional components or overriding shouldComponentUpdate for class components, ensuring they only re-render when necessary.
6. **Context Value Changes**: If the value provided by a React context changes frequently, it triggers re-renders of all components consuming that context, even if they don't need the updated value. We can minimize this by limiting how often the context value changes or by memoizing the context value.
7. **What are the disadvantages of using inline function?**

Using inline functions (like arrow functions) in JSX causes them to be recreated on every render. This leads to unnecessary re-renders in child components if the function is passed as a prop.

* **Disadvantage**: Creates new function references every time, causing performance issues, especially in deeply nested components.
* **Solution**: Use useCallback to memoize functions and prevent them from being recreated on each render.

Explanation:

When you pass an inline function (like an arrow function) as a prop to a child component, React sees this function as a **new reference** on every render, even if the logic inside the function hasn’t changed. This causes the child component to re-render, even if its actual props haven’t changed.

1. How does React.memo works and when should you avoid it ?

**React.memo** is a higher-order component that **memoizes** a functional component's output. It prevents re-renders by performing a **shallow comparison** of the props. If the props haven’t changed, React skips the re-render and reuses the previous render.

**When to avoid React.memo:**

1. **Small, simple components**: For lightweight components that render quickly, the overhead of React.memo may not provide a performance boost.
2. **Frequent prop changes**: If props are complex (objects or arrays) and change often, React.memo will still trigger re-renders.
3. **Deeply nested objects/arrays**: Since React.memo does a shallow comparison, it won’t detect deep changes, leading to unnecessary re-renders.
4. **Components with internal state or side effects**: If a component relies on state or effects, React.memo won’t prevent re-renders triggered by those.
5. **What does useCallback actually do?**

useCallback is a React hook that **memoizes** a function, preventing it from being recreated on every render. It returns a **memoized version** of the callback function that only changes if the dependencies you specify in the dependency array change.

**How it Works:**

1. When you pass a function to a component, React will recreate that function every time the component renders, causing unnecessary re-renders of child components if the function is passed as a prop.
2. useCallback ensures the function is only recreated if one of its **dependencies** has changed, not on every render.

**5) How can use Effect causes performance issues?**

useEffect can lead to performance issues if misused. The common causes are:

1. **Unnecessary Re-renders**:
   * If the **dependency array** is omitted or incorrect, useEffect runs on every render, causing redundant operations.
   * **Solution**: Always use a correct dependency array to control when the effect should run.
2. **Excessive Re-renders from State Updates**:
   * Updating state inside useEffect without proper conditions can cause an **infinite loop** of re-renders.
   * **Solution**: Ensure state updates do not directly trigger the effect to rerun.
3. **Heavy Operations**:
   * Expensive computations or API calls inside useEffect can block the main thread and affect UI responsiveness.
   * **Solution**: Use debouncing/throttling for frequent changes or move heavy computations outside useEffect.
4. **Too Many Dependencies**:
   * Including too many dependencies in the array causes frequent re-renders.
   * **Solution**: Only include essential dependencies in the array.
5. **Not Cleaning Up**:
   * Side effects like event listeners or timers that are not cleaned up can cause **memory leaks**.
   * **Solution**: Always return a cleanup function to clear side effects.

By using useEffect correctly and efficiently, you can avoid these performance pitfalls in React.

Bundling and lazy loading:

**What is Tree Shaking ?**

**Tree shaking** is a term used in **JavaScript bundling** to describe the process of **removing unused code** from your final bundle. It’s a form of **dead code elimination** that helps reduce the size of your application by ensuring that only the **necessary code** is included in the final build.

**How Tree Shaking Works:**

* **Tree shaking** is performed by modern JavaScript bundlers like **Webpack**, **Rollup**, or **Parcel**.
* It works by analyzing **ES6 module imports/exports**. Since ES6 modules have **static structure**, bundlers can determine which code is actually being used and remove the unused parts.
* It "shakes" the code like a tree and drops the "dead" branches (unused code).

**How would you load only critical code above the fold?**

To optimize page load times, you can use the following techniques:

1. **Code Splitting (Lazy Loading)**:
   * Load only critical code initially, and lazy-load non-essential code after the page loads using tools like **React.lazy()**, **Webpack**, or **Rollup**.
2. **Preload and Prefetch**:
   * **Preload** essential resources (CSS, JavaScript, fonts) using <link rel="preload"> to speed up loading.
   * **Prefetch** non-critical resources that might be used later with <link rel="prefetch">.
3. **Inline Critical CSS**:
   * Inline only the CSS needed for above-the-fold content and load the rest asynchronously.
4. **Defer Non-Critical JavaScript**:
   * Use defer or async attributes to load non-essential JavaScript after the page has rendered.
5. **Server-Side Rendering (SSR) or Static Site Generation (SSG)**:
   * Pre-render content on the server to serve a fully rendered page, reducing load time for above-the-fold content.

**List and virtualization:**

**How would you optimize a list of 10000 items?**

To optimize a list of 10,000 items, I would focus on improving **performance**, **memory usage**, and **rendering efficiency**. Here are some strategies:

1. **Pagination/Infinite Scrolling**:
   * Instead of rendering all items at once, load a subset (e.g., 100 items at a time) and use **pagination** or **infinite scrolling** to load more items as needed.
2. **Code Splitting & Lazy Loading**:
   * Split the list into smaller chunks and only load **critical items** above the fold first, using **lazy loading** for the rest.
3. **Virtualization**:
   * Use libraries like **react-virtualized** or **react-window** to render only the visible items, greatly improving performance by reducing the number of DOM elements.
4. **Efficient Algorithms**:
   * Use optimized algorithms for operations like **searching**, **sorting**, and **filtering** (e.g., **binary search**, **merge sort**, etc.) to handle the list efficiently.
5. **Debouncing/Throttling**:
   * If performing actions like search or filter based on user input, use **debouncing** or **throttling** to reduce the number of operations.
6. **Asynchronous Processing**:
   * For heavy computations or sorting, use **async operations** or **Web Workers** to keep the UI responsive.
7. **Memory Optimization**:
   * Use **Typed Arrays** for numeric data and avoid duplicating data to reduce memory usage.

What is the difference between the Windowing and Infinite Scrolling ?

**Windowing** and **infinite scrolling** are related but not the same concept. Here's how they differ:

**Windowing vs Infinite Scrolling:**

1. **Windowing**:
   * **Definition**: Windowing is the technique of rendering only the items visible in the viewport (and a small buffer around them) at any given time. It is used to efficiently render large lists by limiting the number of DOM nodes created.
   * **How it works**: As the user scrolls, the items in the viewport are dynamically updated, while off-screen items are removed from the DOM.
   * **When to use**: Windowing is typically used when you want to display **all items at once** but only render those that are visible to the user. It’s ideal for cases where the full dataset needs to be loaded into the browser but only a subset needs to be displayed.
   * **Libraries**: react-window, react-virtualized.

**Example**:

* + If you have 10,000 items in a list, only a small subset (say, 20-50 items) are rendered at any given time based on the scroll position.

1. **Infinite Scrolling**:
   * **Definition**: Infinite scrolling is a technique where new content is **loaded on-demand** as the user scrolls down a page, typically via API calls or appending more items to the list dynamically.
   * **How it works**: When the user scrolls to the bottom (or near it), the system triggers a fetch request to load more data and appends it to the list.
   * **When to use**: Infinite scrolling is ideal when you have **pagination** and need to load more data dynamically as the user interacts with the list.
   * **Libraries**: react-infinite-scroll-component, react-virtualized.

**Example**:

* + Only a few items (e.g., 20-50) are initially loaded, and as the user scrolls to the bottom, more items are fetched and added to the list.

**Key Differences:**

* **Data Loading**:
  + **Windowing**: The entire dataset is loaded, but only a small portion is rendered at once.
  + **Infinite Scrolling**: Data is **fetched dynamically** as the user scrolls, and the list grows over time.
* **Performance**:
  + **Windowing**: Improves performance by reducing the number of DOM elements rendered at a time, regardless of the total data size.
  + **Infinite Scrolling**: Helps manage large datasets by only fetching the data that is needed, but still requires rendering a larger number of items as the user scrolls.

Profiling and debugging:

What kind of insights can you get from flamegraphs?

How would you debug slow first paint or layout shift?

**Internal and advance behaviour:**

**What is useOptimistic and useActionState in react 19 and improve ux?**

useOptimistic is a React Hook that lets you optimistically update the UI.

**useOptimistic** is a React hook that allows you to **optimistically update the UI** during an asynchronous action, such as a network request. This enables you to **immediately reflect changes in the UI** without waiting for the action to complete, improving the **perceived performance** and responsiveness of the application.

This state is called the “optimistic” state because it is usually used to immediately present the user with the result of performing an action, even though the action actually takes time to complete.

The useOptimistic Hook provides a way to optimistically update the user interface before a background operation, like a network request, completes. In the context of forms, this technique helps to make apps feel more responsive. When a user submits a form, instead of waiting for the server’s response to reflect the changes, the interface is immediately updated with the expected outcome.

**useActionState:**

useActionState is another **experimental hook** that provides a way to manage the state of **asynchronous actions** in a more declarative manner. This hook can be used to track the **pending, success, or error states** of actions like submitting a form, sending a request, or performing any asynchronous task.

It allows you to manage multiple states related to an action (like **loading**, **success**, and **error**) and improves the UX by allowing the application to provide **feedback** to users during the execution of asynchronous tasks.

**How it Works:**

* It provides a **structured way to handle different states** of an action (e.g., loading, error, success).
* It allows the UI to reflect the status of the action in real-time, providing visual feedback like **spinners**, **disabled buttons**, or **error messages**.

What is the difference between the useDefferedValue and useTransition?

 **useTransition** is used to **defer non-urgent updates** (e.g., large data processing, filtering) while ensuring **immediate feedback** for critical tasks (e.g., user input).

 **useDeferredValue** is used to **defer rendering of frequent state updates** to prevent janky UI during rapid user interactions, like typing.

**useDeferredValue** is a **React hook** that allows you to **defer** the rendering of a state update until the browser is less busy, improving UI responsiveness. This hook is commonly used for scenarios where frequent updates, such as text input or search queries, might lead to performance issues due to re-rendering the entire UI for each change.

**Smooth User Experience**: By deferring non-urgent updates, React can prioritize more important updates, ensuring that the UI remains responsive even with frequent state changes

**Use Case:**

useDeferredValue is particularly useful for use cases like **search bars** or **input fields**, where the user is typing rapidly and the UI might need to update frequently.

**How It Works:**

When you use useDeferredValue, React will update the value of the state, but the UI will only be updated at a time when the browser is not heavily occupied with other tasks.

**useTransition:**

**useTransition** is a React hook that allows you to **mark certain updates as non-urgent** or "transitioning" so that they can be deferred without blocking the more urgent updates. This is particularly useful for **slow rendering operations**, such as filtering large lists or navigating between pages, without blocking user interactions like typing or clicking.

* **Purpose**: It allows you to prioritize immediate updates (e.g., user input) while deferring less critical updates (e.g., UI transitions, rendering large lists).
* **How it works**: useTransition returns an array with two items:
  + A **startTransition function** to mark updates that are non-urgent.
  + A **boolean flag (isPending)** that indicates whether the transition is still in progress.

**How it Works:**

When you mark a state update as a transition, React will continue rendering immediate updates (e.g., responding to user input) while deferring the non-urgent updates (e.g., rendering a list of search results).

**What is the useFormStatus ?**

useFormStatus is a Hook that gives you status information of the last form submission.

**What is useMemo and useCallback?**

* **useMemo**: Optimizes performance by memoizing the result of an expensive function.
* **useCallback**: Memoizes a function so that it’s not recreated on every render, useful for passing callbacks to child components.

**Explain the concept of "lifting state up" in React.**

* Lifting state up refers to moving state from a child component to a common parent so that multiple components can share and modify that state.

State lifting in React refers to moving the state from a child component to a parent component in order to share the state or allow multiple components to interact with the same data. Here's a simple example to demonstrate the concept of state lifting:

**Parent components:**

import React, { useState } from 'react';

import ChildComponentA from './ChildComponentA';

import ChildComponentB from './ChildComponentB';

function ParentComponent() {

const [sharedState, setSharedState] = useState('');

const handleChange = (event) => {

setSharedState(event.target.value);

};

return (

<div>

<h2>State Lifting Example</h2>

<ChildComponentA sharedState={sharedState} onChange={handleChange} />

<ChildComponentB sharedState={sharedState} />

</div>

);

}

export default ParentComponent;

**Child Component:**

import React from 'react';

function ChildComponentA({ sharedState, onChange }) {

return (

<div>

<h3>Child A</h3>

<input

type="text"

value={sharedState}

onChange={onChange}

placeholder="Type something"

/>

</div>

);

}

export default ChildComponentA;

What changed in react18 with concurrent rendering?

Situational Based react coding Question:

**You need to create a form that dynamically handles multiple sections (like address, payment, contact information). How would you implement this in React?**

* **What the interviewer is looking for**: Understanding of **controlled components**, **state management** (using React state), and dynamic form handling.

**Example Answer**:

* You can manage each section's state using useState for form data. Use a dynamic rendering approach where you switch between sections based on user navigation.

js

Copy

const [formData, setFormData] = useState({

address: '',

payment: '',

contact: ''

});

const handleChange = (e) => {

const { name, value } = e.target;

setFormData({

...formData,

[name]: value

});

};