1. React functional component lifecycle: mounting > updating > unmounting
   1. Only called when component is mounted:   
      useEffect(() {  
      // logic  
      }, [])
   2. Only called when component is mounted and updated:   
      useEffect(() {  
      // logic  
      })
   3. Called when component is mounted and unmounted(return statement):   
      useEffect(() {  
      // logic  
      return() => {  
      // logic  
      }  
      }, [])
2. **Controlled vs uncontrolled components**
   1. A [Controlled Component](https://facebook.github.io/react/docs/forms.html#controlled-components) is one that takes its current value through props and notifies changes through callbacks like onChange. A parent component "controls" it by handling the callback and managing its own state and passing the new values as props to the controlled component. You could also call this a "dumb component".
   2. A [Uncontrolled Component](https://facebook.github.io/react/docs/uncontrolled-components.html) is one that stores its own state internally, and you query the DOM using a ref to find its current value when you need it. This is a bit more like traditional HTML.
3. **Props vs state**
   1. **Props** get passed to the component similar to function parameters
   2. **State** is managed within the component similar to variables declared within a function
4. **Phases of reactJS lifecycle**
   1. **Initialization:** In this phase react component prepares setting up the initial state and default props.
   2. **Mounting:** The react component is ready to mount in the browser DOM. This phase covers **componentWillMount** and **componentDidMount** lifecycle methods.
   3. **Updating:** In this phase, the component get updated in two ways, sending the new props and updating the state. This phase covers **shouldComponentUpdate, componentWillUpdate and componentDidUpdate** lifecycle methods.
   4. **Unmounting:** In this last phase, the component is not needed and get unmounted from the browser DOM. This phase include **componentWillUnmount** lifecycle method.
5. **Lifecycle methods of react (Class based Components)**
   1. **componentWillMount:** Executed before rendering and is used for App level configuration in your root component.
   2. **componentDidMount:** Executed after first rendering and here all AJAX requests, DOM or state updates, and set up eventListeners should occur.
   3. **componentWillReceiveProps:** Executed when particular prop updates to trigger state transitions.
   4. **shouldComponentUpdate:** Determines if the component will be updated or not. By default it returns true. If you are sure that the component doesn't need to render after state or props are updated, you can return false value. It is a great place to improve performance as it allows you to prevent a rerender if component receives new prop.
   5. **componentWillUpdate:** Executed before re-rendering the component when there are pros & state changes confirmed by shouldComponentUpdate which returns true.
   6. **componentDidUpdate:** Mostly it is used to update the DOM in response to prop or state changes. Takes in two input parameters (**prevProps**, **prevState**)
   7. **componentWillUnmount:** It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.
6. DOM, Virtual DOM, Shadow DOM
   1. DOM: API for HTML and XML documents. Creates a logical structure which can be accessed and manipulated by browsers
   2. Virtual DOM: Creates copy of entire DOM and stores in memory. On change of component, it updates Virtual DOM. Virtual DOM is compared with DOM, if any changes they are reflected in DOM.
   3. Shadow DOM: Creates small pieces of DOM which has their own isolated scope for the element they represent. Eg: video tag is html. DOM doesn’t shows volume or play/pause button
7. When component renders
   1. When the state changes of component
   2. When the value of props which the component receives changes
   3. When parent re-renders, by default the child also renders
8. Memoization
   1. Speed optimization technique where given a functions, it returns the cached version of output for same inputs
   2. It remembers output for the given set of inputs
   3. It does a shallow comparison (reference check for non-primitive(object, arrays or functions) that memory location is still same or not) of props, if value changed, it re-renders
9. useMemo, memo and useCallback
   1. React.memo – similar to React.purecomponent() for class based components. It will compare all props passed to the component by referential equality. If these props are unchanged, React.memo will reuse the last rendered result, therefore, it prevents the component from being re-rendered.
   2. useMemo and useCallback: Both React.useMemo and React.useCallback receives a function as its first argument and a dependencies array as the second one. The hook will return a new value only when one of the dependencies value changes (referential equality). The main difference is that React.useMemo will call the function which is received and return its result while React.useCallback will return the received function without calling it.
   3. useMemo – memorizes the result of the function that is called. use React.useMemo when we compute expensive value that we don't want to compute it again and again when the component is rerendered
   4. useCallback – memorizes a function which can later be called
10. React vs Node Js: <https://www.geeksforgeeks.org/difference-between-node-js-and-react-js/>
11. Redux vs Context API :
12. React Hooks :
    1. useState - useState is a Hook that lets you add React state to function components
    2. useEffect – single hook to take care of **componentDidMount and componentDidUpdate. R** **uns both after the first render and after every update**
    3. useReducer - To help separate the concerns (rendering and state management) React provides the hook useReducer(). The hook does so by extracting the state management out of the component.
    4. useRef -
    5. createContext – useContext  
         
       <https://youtu.be/o-alRbk_zP0>
    6. useCallback - Pass an inline callback and an array of dependencies. useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders
    7. useMemo - useMemo will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.
    8. React.memo() - it does a shallow comparison. It works good with strings and numbers. Doesn’t works when passed in objects or functions (useCallback/useMemo helps) . It will compare all props passed to the component by referential equality. If these props are unchanged, React.memo will reuse the last rendered result, therefore, it prevents the component from being re-rendered
13. Stopwatch in react
14. List in react - <https://www.javatpoint.com/react-lists>
15. Dynamic form - <https://www.freecodecamp.org/news/build-dynamic-forms-in-react/>
16. Higher order Component
17. React Routers – Browser, Memory, Hash (<https://learnwithparam.com/blog/different-types-of-router-in-react-router/>) <https://codesandbox.io/s/objective-nash-illdyn?file=/src/App.js>
18. React Pure Component – A React component is said to be pure if it renders the same output for the same state and props. For React **pure class components**, React provides the **PureComponent** base class. Class components that extend the React.PureComponent classes are treated as pure components. To create a pure functional component in React, React provides a React.memo() API. Using the React.memo() API, the React functional component can be wrapped as follows to get React Pure Functional Component.
19. <React.StrictMode/> -
    1. Identifying components with unsafe lifecycles
    2. Warning about legacy string ref API usage
    3. Warning about deprecated findDOMNode usage
    4. Detecting unexpected side effects
    5. Detecting legacy context API
20. Reconciliation: Reconciliation is the process through which React updates the Browser DOM. It uses virtual DOM approach. Refer to point 6
21. **Babel** - Babel is a transpiler i.e. it converts the JSX to vanilla JavaScript. You can view babel as an intermediate step between your code and "executable" code.
22. **Prop drilling** –
23. **React Flux Architecture**
24. **State lifting**