what do you expect if we use debounce technique with the search input of the above project?

Debouncing is a optimization technique to limit the amount of times a task occurs. If you’ve ever implemented a search feature in React that automatically filters a list or sends a fetch request while a user inputs every character, this is a technique that can greatly improve the efficiency of your application.

**Setup / Installation**

For this example we’ll be importing the debounce function from the lodash utility library instead of creating our own version. To do this we’ll add this line of code to the top of the component that will handle the filtering operation.

In the CodeSandbox example above I have gone ahead and created a simple app that displays a list of fruit and a search bar that will filter the list as you type into it.

The search bar is a [Controlled Form](https://reactjs.org/docs/forms.html). The value it displays is taken from state and the input is linked with state through its onChange synthetic event.

**The Problem**

While this example works fine, you can imagine as the size of the list grows, the longer each search takes. Another scenario could be that we’re sending api requests for each character being searched. Either way this implementation can start to slow the application and you’ll be left wondering how to fix this. In comes debounce .

**Debounce**

\_.debounce(func, [wait=0], [options={}])

While debounce takes 3 arguments, we’ll be using the first two, the function we want to be invoked and the time we’re setting to wait.

We’ll also use the .cancel function that’s built into debounce to help with cleanup (more on that later).

For more info on the debounce function refer to the [official documentation](https://lodash.com/docs/4.17.15#debounce).

**Implementation**

The first step will be to work with useMemo to memoize a return value from our debounce function. This returned value will persist between re-renders. This step is essential because if we don’t persist this data between re-renders other implementations of debounce will occur on every re-render and we would essentially have our original example; We’d be filtering a list per character instead after a set time spent after the last character input.

const debouncedResults = useMemo(() => { return debouce(handleChange, 300);}, []);

Next we’ll work with useEffect to clean up any side effects from debounce when our component gets unmounted; There’s no reason for a search to run when we’re not on that page or view anymore. This is where we’ll call the .cancel function on our memoized return value.

useEffect(() => { return () => { debouncedResults.cancel(); };});