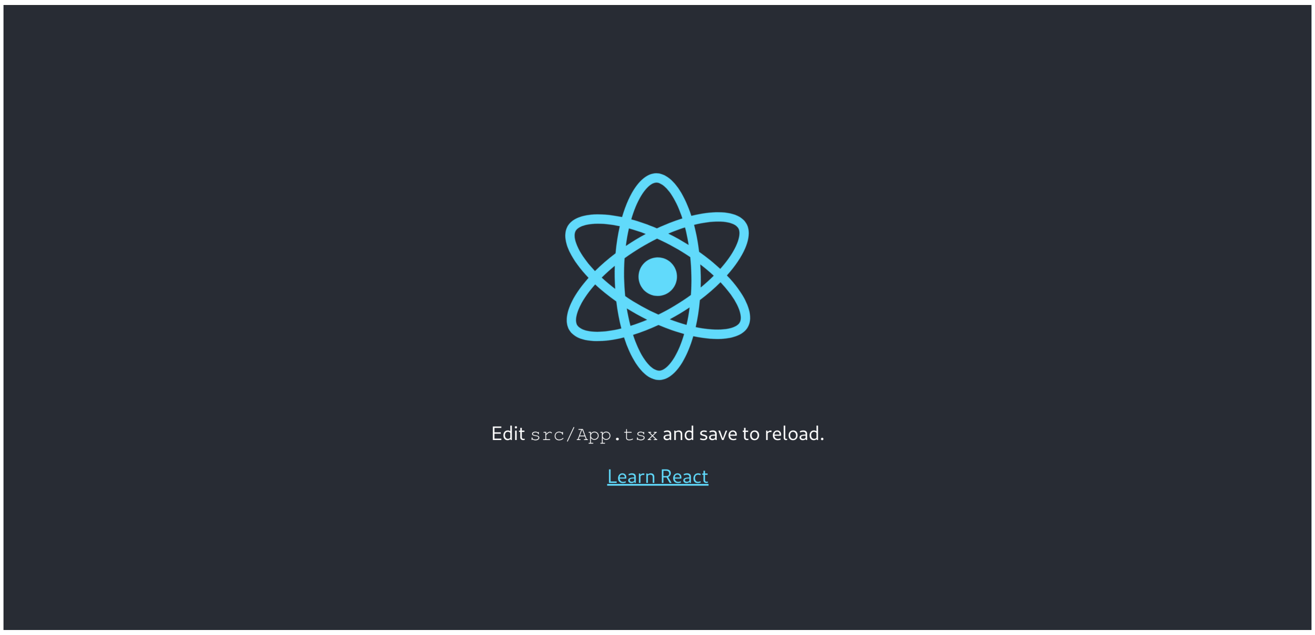
EPQ

Setup

To start off with, I create a typescript react app with the command `npx create-react-app artefact --template typescript`. This creates a folder called artefact containing all starting code for the project. This folder contains some boilerplate code, most of which will be removed. running the command `npm start` starts the boilerplate app, which looks like this:



Edit `src/App.tsx` and save to reload.

Learn React

The artefact folder looks like this:

```
artefact
├── node_modules
├── package.json
├── package-lock.json
├── public
├── README.md
├── src
└── tsconfig.json
```

`node_modules` contains the required modules for the project. `Public` contains the `index.html` document where the react app will be injected. And `src` is where the typescript files for the app will go.

I start off by removing all the boilerplate code from the src directory. This leaves me with four files,

```
artefact/src
├── App.css
├── App.tsx
├── index.css
└── index.tsx
```

`App.css` is empty and will be where all the component's CSS will go
`App.tsx` is the typescript file where the react components will go. Currently, it contains:

```
import React from "react";
import "./App.css";

function App() {
    return <div className="App"></div>;
}

export default App;
```

This code returns an empty div element to be injected into the app.

`index.css` contains some extra CSS for index.html, currently it just contains some font styling:

```
body {
    margin: 0;
    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", "Roboto",
        "Oxygen", "Ubuntu", "Cantarell", "Fira Sans", "Droid Sans",
        "Helvetica Neue", sans-serif;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
}

code {
    font-family: source-code-pro, Menlo, Monaco, Consolas, "Courier New",
        monospace;
}
```

`index.tsx` contains code that injects the empty app element created in `App.tsx` :

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";

ReactDOM.render(
    <React.StrictMode>
        <App />
    </React.StrictMode>,
    document.getElementById("root")
);
```

Sorting

Basic page setup

The sorting algorithm page will contain these elements:

- The algorithm visuliser
- A select element to select the selected algorithm
- Buttons to start, stop and reset the visualizer

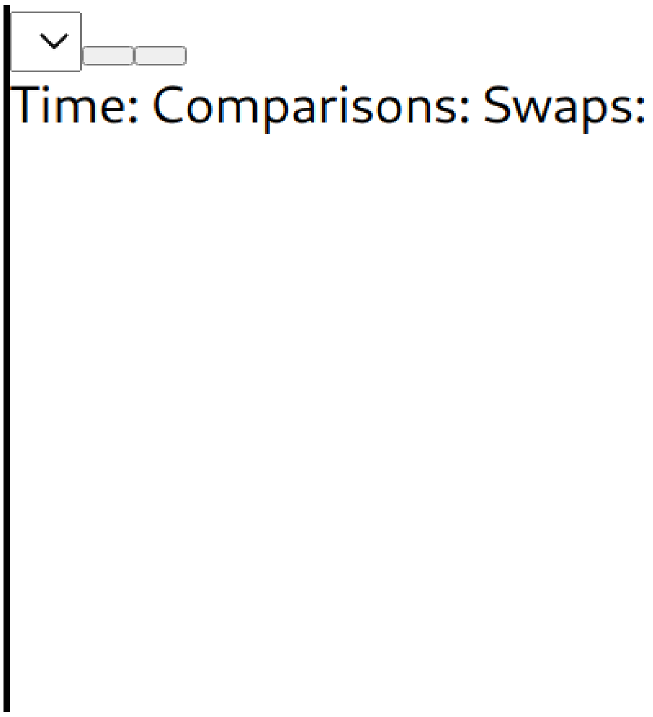
- Scrolling slide bars to select the number of bars to be sorted, and the sorting speed
- A panel containing metrics on the algorithm, which will display the time taken, number of comparisons and number of swaps
- A place to describe the selected algorithm

So to start off with, I need to create these components inside `App.tsx`. I have not decided on a final visual design for the application, so I will not style the components any more than required to get them working. Then, once I have decided on how the application will look, I will add the CSS to style the components. The file `App.css` now looks like this:

```
import React from "react";
import "./App.css";

// React element for the bar container
const BarContainer = () => {
    return <div className="barContainer"></div>;
};

// React element for the Controlls
const Controlls = () => {
    return (
        <div className="controlls">
            <select className="algorithmSelect"></select>
            <button className="startstop"></button>
            <button className="reset"></button>
            <input type="range" />
            <input type="range" />
        </div>
    );
};

// React element for the Metrics
const Metrics = () => {
    return (
        <div className="metrics">
            <span>Time: </span>
            <br />
            <span>Comparisons: </span>
            <br />
            <span>Swaps: </span>
        </div>
    );
};

// react element for the Description
const Description = () => {
    return <div className="description"></div>;
};

// Base app
function App() {
    return (
        <div className="App">
            <BarContainer />
            <Controlls />
            <Metrics />
            <Description />
        </div>
    );
}

export default App;
```

'App.tsx` now contains empty components for the elements of the page. I decided to put the speed, swaps, and comparisons in their own components to make things easier later when I have to update them with their values dynamically.

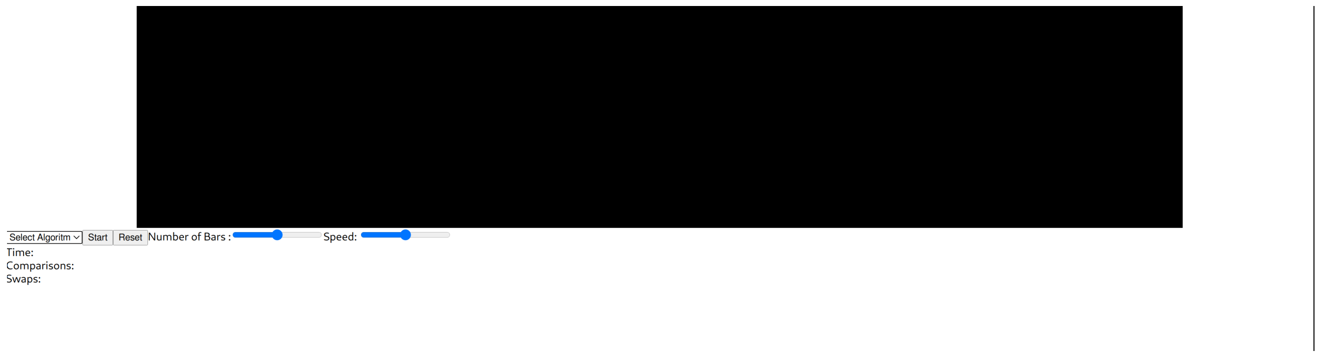
Now when I run the app it looks like this:


Time: Comparisons: Swaps:

Because there's no styling at all, it is very hard to tell what is going on. Therefore I added some basic CSS styling for the bar container inside `App.css` :

```
.barContainer {
    width: 80vw;
    height: 20rem;
    background: black;
    margin: auto;
}
```

This, plus some added text inside the controls, makes the app now look like this:



Time:
Comparisons:
Swaps:

Adding Bars

Now to add some bars, I started by by creating an interface and functional component for the bar:

```
//interface for the props passed into the bar
interface barProps {
    size: number;
    maxSize: number;
    key: number;
}
// React element for the bars
const Bar = (props: barProps) => {
    // CSS styles for bar
    var style: CSSProperties = {
        height: ((props.size / props.maxSize) * 100).toString() + "%",
    };
    return <div className="bar" style={style}></div>;
};
```

The interface declares the type of props being passed into the bar. These props are then used to determine the height of the bar. The height of a bar is represented as $\left(\frac{BarSize}{MaxBarSize} \times 100\right)$% of the bar containers height.

To create the bars, I first needed a way of creating and storing the size of each bar. I achieved this by creating a `bars` array inside the program state. This array will contain the size of each bar. I also added a variable, `numOBars`, which contains the number of bars to be visualised.

I then created a method that adds the numbers 1 -> numOBars to the bars array. This method is then run when the page loads to create the bars.

```
// Base app
class App extends React.Component {
    componentDidMount() { // runs once the component has been loaded
        this.makeBars();
    }

    state = {
        numOBars: 50,
        bars: [],
    };

    // method to make the bars
    // if no parameters are entered, n is taken to be the number of bars
    makeBars = (n = this.state.numOBars) => {
        var b = [];
        for (let i = 0; i < n; i++) {
            // create an array b containing 1 -> n
            b.push(i + 1);
        }
        this.setState({ bars: b, numOBars: n }); // sets the state of bars to be
b
    };
```

These state values are then passed into the barContainer component, where they are used to create the bars.

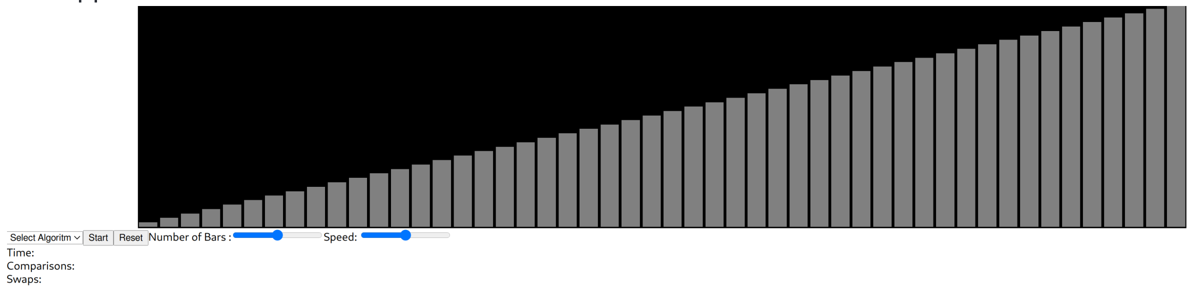
```
//interface for the props passed into the bar container
interface barContainerProps {
    bars: Array<any>;
    maxSize: number;
}

// React element for the bar container
const BarContainer = (props: barContainerProps) => {
    return (
        <div className="barContainer">
            {props.bars.map(
                (
                    bar //loop through all the array, creating a bar for each
element
                ) => (
                    <Bar size={bar} maxSize={props.maxSize} key={bar} />
                )
            )}
        </div>
    );
};
```

To make the bars centred and visible, I added this CSS styling

```
.barContainer {
    width: 80vw;
    height: 20rem;
    background: black;
    margin: auto;
    display: flex;
    align-items: flex-end;
    justify-content: center;
}

.bar {
    background: gray;
    width: 5rem;
    margin: 2px;
}
```

I'm using a CSS flexbox for the bar container for two reasons. Firstly It will allow me centrally align the bars. Secondly, it will compress the elements inside of it so that they all fit. This means that all the bars will dynamically adjust their widths based on how much space is available.

The app with the bars now looks like this:



The next thing to do was to allow the number of bars to be edited by the range input created earlier. I moved the code for the `Controls` component into its own file named 'controls.tsx`. This created the codebase easier to deal with as the code is spread over multiple files.

Inside that file I added this code:

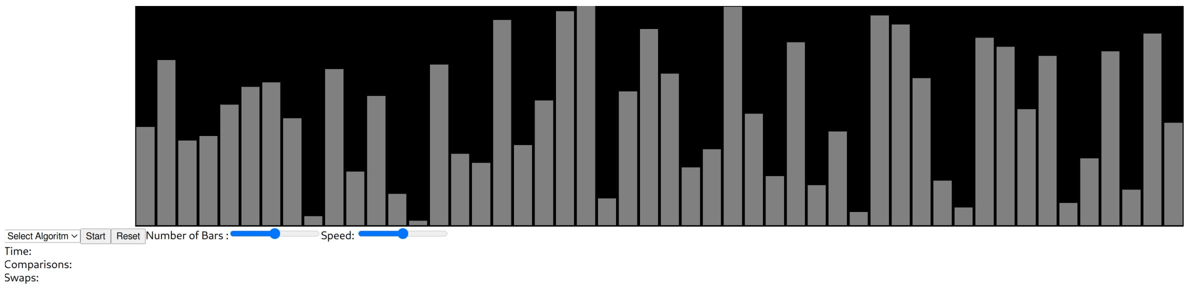
```
import React from "react";
import "./App.css";

interface controlProps {
    makeBars: any;
}

// react component for the control pannel
export default class Controls extends React.Component<controlProps> {
    barSelect: React.RefObject<HTMLInputElement>;
    constructor(props: any) {
        super(props);
        this.barSelect = React.createRef(); // creates a ref which will be
assigned to the bar select element
    }

    // calls the makeBars method from the app class and passes in the value of
the length range
    makeBars = () => {
        this.props.makeBars(this.barSelect.current?.value);
    };

    public render() {
        return (
            <div className="controls">
                <select className="algorithmSelect">
                    <option value="">Select Algoritm</option>
                </select>
                <button className="startstop">Start</button>
                <button className="reset">Reset</button>
                Number of Bars :
                <input
                    type="range"
                    ref={this.barSelect} // linking the barSelect ref to the
element
                    onChange={() => this.makeBars()} // call the `makeBars`
method whenever the value of the range is changed
                />
                Speed: <input type="range" />
            </div>
        );
    }
}
```

Whenever the value of the range representing the number of bars changes, the method `MakeBars` is called, and the bars are remade.

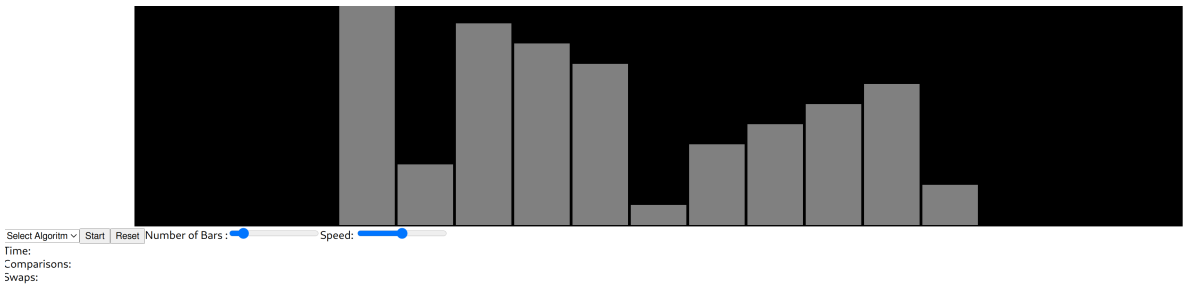
At the moment, the bars are always created in order, which makes sorting them pretty boring as they are already in order. Therefore inside the `MakeBars` method, I added some code to shuffle the array.

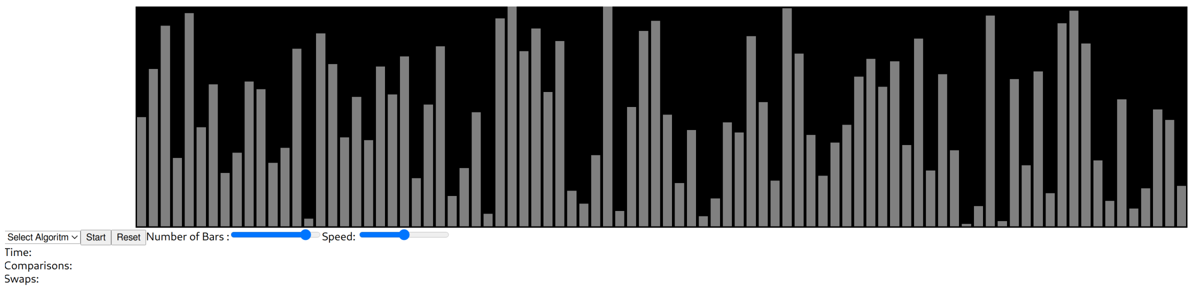
```
// shuffles array
for (let i = b.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1));
    [b[i], b[j]] = [b[j], b[i]];
}
```

Now, the bars look like this:



And the range slider can be adjusted to increase or decrease the number of bars, which can be seen here:





Algorithm select

The way this application works will be whenever an algorithm is selected, it is run with the state of the bars being appended to the end of an array for every step. Then to display the visualisation of the algorithm, that array is iterated through, with each iteration being set as the bars' current state. I am doing it this way rather than sorting them live (changing the program state as the sorting algorithm is running) because it will allow the user to pause and then resume the visualisation with ease. It will also enable the ability to step forwards and backwards through visualisation.

To start off with, I added three things to the application state, an array containing all the algorithms available to be visualised, an array for the sorting stages, and the stage at which the visualisation is on.

```
state = {
    numOBars: 50,
    bars: [],
    algorithms: [
        "Bubble Sort",
        "Quick Sort",
        "Insertion Sort",
        "Merge Sort",
        "Selection Sort",
        "Heap Sort",
        "Radix Sort",
        "Bucket Sort",
    ],
    sortingStages: [],
    sortingStage: 0,
};
```

Next, inside the Controls component, I modified the element which selects the algorithm to be used:

```
<select
    className="algorithmSelect"
    onInput={() ⇒ this.algorthmSelected()}
    ref={this.algorithmSelect}
>
    <option value="">Select Algoritm</option>
    {this.props.algorithms.map((a) ⇒ (
        <option value={a} key={this.props.algorithms.indexOf(a)}>
            {a}
        </option>
    ))}
</select>
```

Now it displays an option for all the algorithms. Next, I created a new file named `algorithms.ts` which is where I will put all the functions for the algorithms. These functions will take in the bars array and return an array containing all the steps involved in sorting the bars. Then, inside the App class, I created a method to run the sorting functions. This method will be passed into the Controls class and run whenever an algorithm is selected. At the moment, it only calls for the bubble sort for testing purposes. Once the code works, I can add in the other algorithms.

```
// method to run whichever sorting algorithm is selected
runAlgorithm = (selected: string) ⇒ {
    var sortingStages;
    switch (
        selected // Switch statement to select the algorithm to use
    ) {
        case "Bubble Sort":
            sortingStages = bubble(this.state.bars);
    }
    this.setState({ sortingStages: sortingStages, sortingStage: 0 }); // sets
the sortingStages array inside the state to be the stages genetrated by the
algorithm, also sets the sorting stage to 0
};
```

I then added the method which the algorithm select calls onInput

```
// method to run whichever sorting algorithm is selected
runAlgorithm = (selected: string) ⇒ {
    var sortingStages;
    switch (
        selected // Switch statement to select the algorithm to use
    ) {
        case "Bubble Sort":
            sortingStages = bubble(this.state.bars);
    }
    this.setState({ sortingStages: sortingStages, sortingStage: 0 }); // sets
the sortingStages array inside the state to be the stages genetrated by the
algorithm, also sets the sorting stage to 0
};
```

Bubble sort

Description

Bubble sort is one of the simplest algorithms to understand and implement. It works by repeatedly iterating through an array and continuously swapping adjacent elements until the list is in order. If you were sorting in ascending order, the algorithm would step through an array and at every step, it checks if the current element is larger than the next element. If it is, those elements are swapped. Once the algorithm reaches the end of an array, it repeats the process, starting from the begging. This results in the largest elements 'bubbling' up the array. Each iteration of the array is known as a run. You know the array is sorted once no elements are swapped in a run. This is because every element is less than the element that follows it.

Complexity

Type	Worst case	Best case	Average
Comparisons	$O(n^2)$	$O(n)$	$O(n^2)$
Swaps	$O(n^2)$	$O(1)$	$O(n^2)$

step by step example

A bubble sort of the array `[4, 3, 7, 2, 6]`

First Pass:

[**4**, **3**, 7, 2, 6] -> [**3**, **4**, 7, 2, 6] the first two elements are swapped as 3 > 4
[3, **4**, **7**, 2, 6] -> [3, **4**, **7**, 2, 6] 7 > 4 so they aren't swapped
[3, 4, **7**, **2**, 6] -> [3, 4, **2**, **7**, 6] 2 < 7 so are swapped
[3, 4, 2, **7**, **6**] -> [3, 4, 2, **6**, **7**] 6 < 7 so are swapped

Second Pass:

[**3**, **4**, 2, 6, 7] -> [**3**, **4**, 2, 6, 7]
[3, **4**, **2**, 6, 7] -> [3, **2**, **4**, 6, 7]2 < 4 so are swapped
[3, 2, **4**, **6**, 7] -> [3, 2, **4**, **6**, 7]

Third Pass:

[**3**, **2**, 4, 6, 7] -> [**2**, **3**, 4, 6, 7]2 < 3 so are swapped
[2, **3**, **4**, 6, 7] -> [2, **3**, **4**, 6, 7]

The array is sorted. However, the algorithm doesn't know that yet. Therefore another pass is needed.

Forth Pass:

[**2**, **3**, 4, 6, 7] -> [**2**, **3**, 4, 6, 7]

Implementation

Here is my implementation of the bubble sort algorithm:

```
export const bubble = (bars: bar[]) => {
    var stages = [];
    stages.push(JSON.parse(JSON.stringify(bars))); // push first stage to array

    var swapped = true;
    for (var n = 0; n < bars.length && swapped; n++) {
        // stop once a pass has completed with no swaps
        swapped = false;
        for (var i = 0; i < bars.length - 1 - n; i++) {
            // loops through the array, with each pass one less element needs to
be checked as you know is in the correct position
            if (bars[i].size > bars[i + 1].size) {
                [bars[i], bars[i + 1]] = [bars[i + 1], bars[i]]; // swaps
elements
                swapped = true;
            }
            stages.push(JSON.parse(JSON.stringify(bars))); // pushes step to
stages
        }
    }

    return stages;
};
```

Now, whenever bubble sort is selected, all the steps for bubble sort are computed and saved. An issue at the moment, though, is that if you change the number of bars or reset them, the steps are not updated. One way I could fix this is to call the `algorithmSelected` function every time the bars are redrawn; however, this would be very computationally heavy, so a better way to do it is this. Store a new variable named `selectedAlgoritm` inside the application state. Then add a new method inside the App class, which sets this value. That method will then be called instead of `runAlgorithm` whenever a new algorithm is selected. `runAlgorithm` will then only run when the user presses play for the first time.

```
// method to set the selected algorithm
setAlgorithm = (algorithm: string) => {
    this.setState({ selectedAlgorithm: algorithm });
};
```

```
// calls the setAlgorithm method inside the app class
algorthmSelected = () => {
    this.props.setAlgorithm(this.algorithmSelect.current?.value);
};
```

Select algorithm fix

Now, whenever bubble sort is selected, all the steps for bubble sort are computed and saved. An issue at the moment, though, is that if you change the number of bars or reset them, the steps are not updated. One way I could fix this is to call the algorithmSelected function every time the bars are redrawn; however, this would be very computationally heavy, so a better way to do it is this. Store a new variable named `selectedAlgoritm` inside the application state. Then add a new method inside the App class, which sets this value. That method will then be called instead of `runAlgorithm` whenever a new algorithm is selected. `runAlgorithm` will then only run when the user presses play for the first time.

```
// method to set the selected algorithm
setAlgorithm = (algorithm: string) => {
    this.setState({
        selectedAlgorithm: algorithm,
        stagesGenerated: false,
        isRunning: false,
    });
};
```

```
/ calls the setAlgorithm method inside the app class
    algorthmSelected = () => {
        this.props.setAlgorithm(this.algorithmSelect.current?.value);
    };
```

Visulisation

First, I added a new variable inside the system class named `isRunning`, which will indicate when the visualiser is running. I then passed that variable to the `Controls` class, where it is used to set what is said on the playPause button.

That button then calls a method inside the App class called `togglePlayState` onClick.

```
// method to toggle the display state of the algorithm
togglePlayState = async () => {
    if (this.state.selectedAlgorithm == "") {
        alert("No algorithm selected");
    } else {
        if (!this.state.stagesGenerated) {
            // generates the stages if not generated
            await this.runAlgorithm();
        }
        await this.setState((prevState: any) => ({
            //wait for the state to be toggled
            isRunning: !prevState.isRunning,
        }));
        this.visulise();
    }
};
```

Firstly this method checks if an algorithm has been selected. If not, an alert is displayed. Then, it generates the sorting stages if they have not already been generated. Following that, it then toggles the `isRunning` variable inside the application state. I have set it to await this function because otherwise, it runs asynchronously, which will result in the variable not being toggled when the visualisation starts, causing it not to start.
I then run the visualisation method, which will visualise The selected algorithm.

```
visulise = async () ⇒ {
    // checks if the visulisation is running and that there are stages left to
visulise
    if (
        this.state.isRunning &&
        this.state.sortingStage < this.state.sortingStages.length
    ) {
        // sets the state to be the next stage of the sorting and increments
sortingStage
        this.setState((prevState: any) ⇒ ({
            bars: prevState.sortingStages[prevState.sortingStage],
            sortingStage: prevState.sortingStage + 1,
        }));
        await pause(100); // delay
        this.visulise(); // Calls itself to keep visulising
    }
};
```

This method is recursive, which means it calls itself. It will continue to call itself until either `this.isRunning` is set to false or all the sorting stages have been visualised.

Speed

The `pause` function pauses the operation of the visualiser for as many milliseconds that are passed in

```
// function for creating a delay
const pause = (time: number) ⇒ {
    return new Promise((resolve) ⇒ setTimeout(resolve, time));
};
```

Now, when bubble sort is selected, and the start button is pressed, bubble sort will start or resume being visualised. Likewise, the visualisation is paused when the stop button is pressed. At the moment, the algorithm has a fixed delay of 100ms hardcoded into the visualise method. Now I need to make it so that the speed range slider affects the delay of the visualisation.

I did this similarly to how the bars are set. Whenever the range slider for the speed is changed, a function is called to set the `speed` variable in the program state to 1000 - range value. that is then used in the place of the hardcoded 100ms.

```
// method to set the speed of the visualiser
setSpeed = (speed: number) ⇒ {
    this.setState({ speed: speed });
};
```

```
// calls the setSpeed method in the App class
setSpeed = () => {
    var speed: any = this.speed.current?.value; // get value from range
    this.props.setSpeed(1000 - speed);
};
```

Now, the speed slider can change the speed at which the visualiser is displayed.