**SwapAnimation**

**Design**

The animation of sorting process depends on a component called SwapAnimation. It is used to animate an array of numbers as bars with different height, representing each number's value. It is designed to provide a smooth animation of swapping elements. Bars will not suddenly swap their position. Instead, this component can smoothly modify bars' colours, x-axis and y-axis values to visualise the comparing, swapping process intuitively. In figure 1, bars 1, 2, 6 and 10 have different y value and colour compared to the right-hand bars.

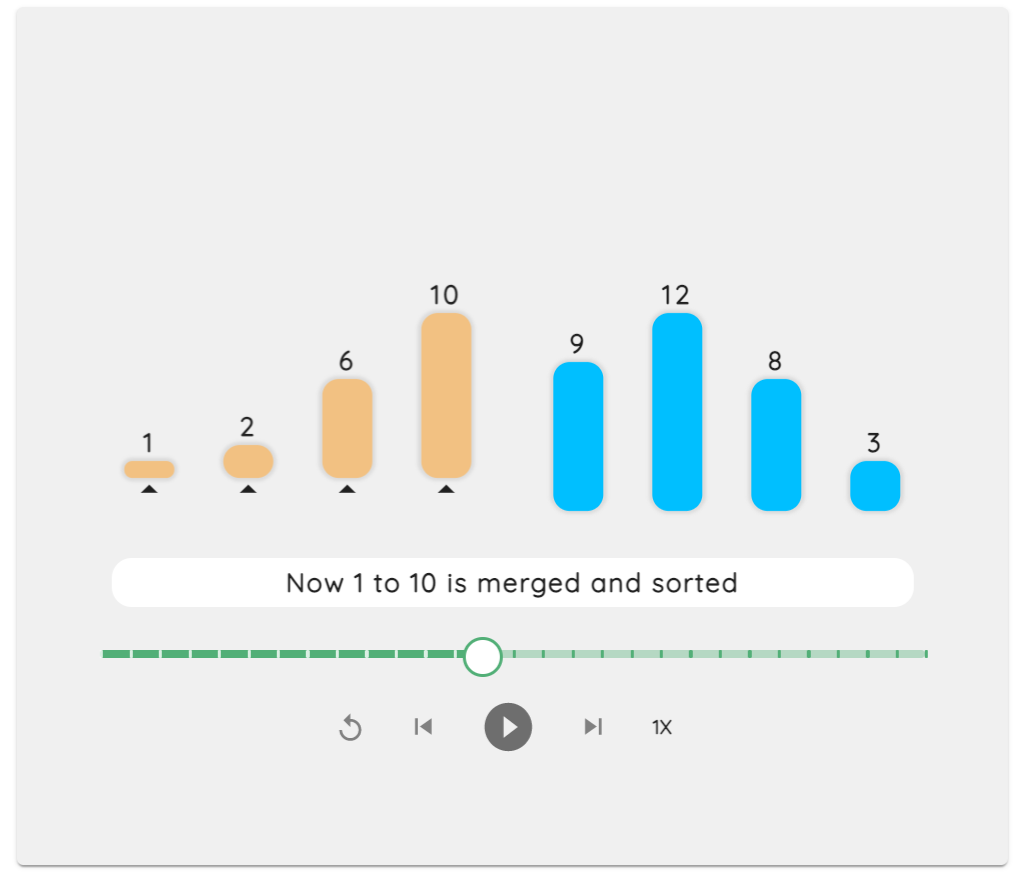


Figure 1. Bars with different colours and y-axis values

SwapAnimation takes a property called trace, an array of bar states in a sequence of time frames. Each element in the trace array contains values and colours of bars and an explanation sentence displayed in the white explanation box in the middle. After each specific time interval, this component will visualise one of a series of bars in order, to illustrate the process of sorting.

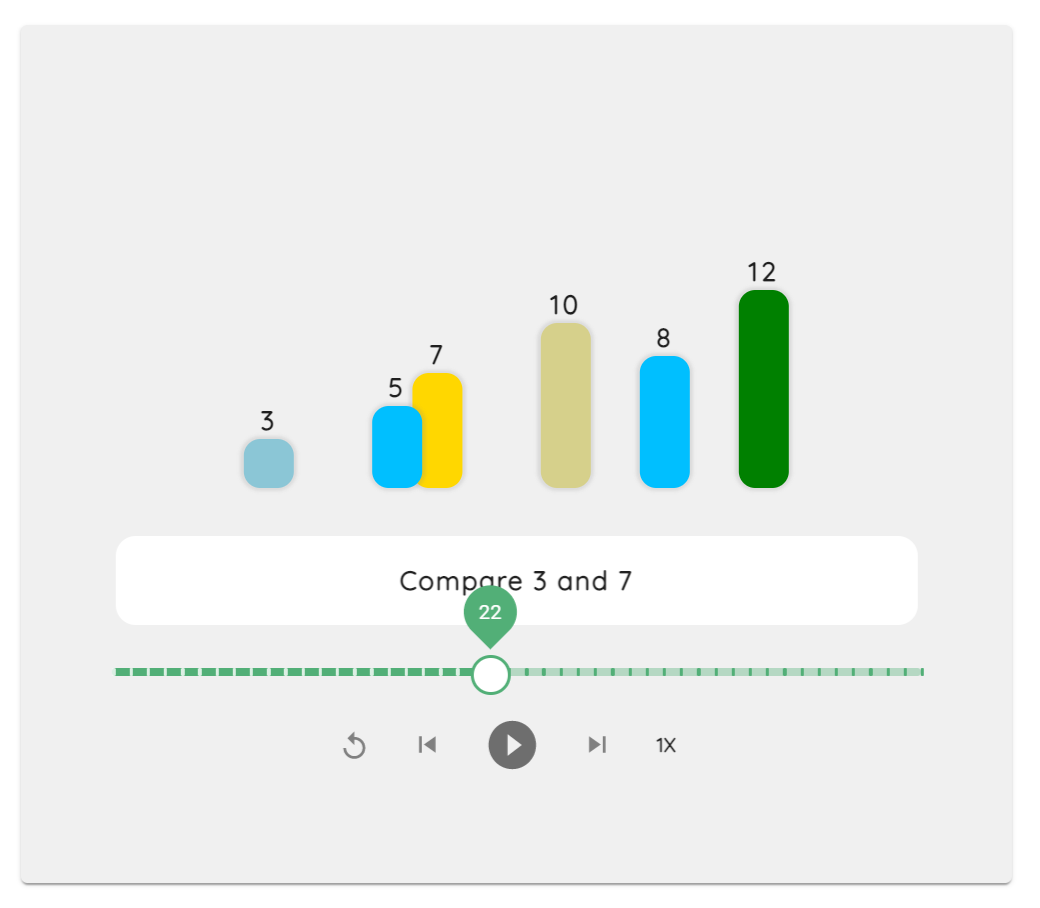


Figure 2. Bars are swapping and changing colours in the step 22

The long green bar with a thumb on it is a progress bar for locating a specific time frame as users want. Five buttons below are reset, step backwards, play/pause and speed.

**Implementation details**

The animation is realised by a component “motion” from a third-party library, Framer Motion [1]. Transition property produces the animation effect, and animate property can adjust background colour and x, y position of a bar.

<motion.li

    key={bar.key} // each bar's identification

    layout

    transition={spring}

    style={bar}

    className={classes.bar}

    animate={{

    backgroundColor: bar. backgroundColor,

    y: bar.y,

    x: bar.x,

    }}

>

Code 1

The timeline of animation is defined by a function called setTimeout. We use each trace's order to set different timeout periods for traces so that bars would be modified after each specific time interval.

subTrace.forEach((item, i) => {

    let timeoutId = setTimeout(

        (item) => {

            // update the current step

            setCurrentStep((prevStep) =>

                i === trace.length - 1 ? prevStep : prevStep + 1

            );

            // update bars to be animated

            setBars(item);

            i === subTrace.length - 1

                ? setIsPlaying(false)

                : setIsPlaying(true);

        },

        i \* timer, //time interval

        item

    );

    timeoutIds.push(timeoutId);

});

Code 2

This component consists of three subparts, AnimationControl, AnimationSlider and InputBar, allowing users to control the animation and define the array of numbers.

**AnimationControl**

This component contains five buttons which are reset, step backwards, play/pause, step forward and speed.

Pause is realised by clear the timeout arguments set in Code 2.

// It is used to clean timeouts to pause the animation

const clearTimeouts = () => {

    timeOutIds.forEach((timeoutId) => clearTimeout(timeoutId));

    setTimeOutIds([]);

};

// To pause the animation

const pause = () => {

    setIsPlaying(false);

    clearTimeouts();

};

Code 3

Play is realised by slice the incomplete traces as a new trace for the component to play.

// To resume the animation

const resume = () => {

    setIsPlaying(true);

    const newtrace = trace.slice(currentStep);

    run(newtrace);

};

Code 4

Reset is realised by pause the animation and set the trace to the initial state.

const handleResetClick = () => {

    pause();

    setCurrentStep(0);

    setBars(trace[0]);

};

Code 5

Step forward and backwards are realised by increasing or decreasing the trace index by one and displaying the updated state.

// Go to next step and pause

const stepForward = () => {

    if (currentStep < trace.length - 1) {

        pause();

        const item = trace[currentStep + 1];

        setCurrentStep((prevStep) => prevStep + 1);

        setBars(item);

    }

};

// Go to the previous step and pause

const stepBackward = () => {

    if (currentStep > 0) {

        pause();

        const item = trace[currentStep - 1];

        setCurrentStep((prevStep) => prevStep - 1);

        setBars(item);

    }

};

Code 6

**AnimationSlider**

AnimationSlider is a progress bar with scales and a thumb on it. Users can drag the thumb to go to a specific time frame of the whole animation. Each time the value of the slider changes, it will update the state of bars.

const handleSliderChange = (event, newValue) => {

    if (isPlaying) {

        pause();

    }

    const item = trace[newValue];

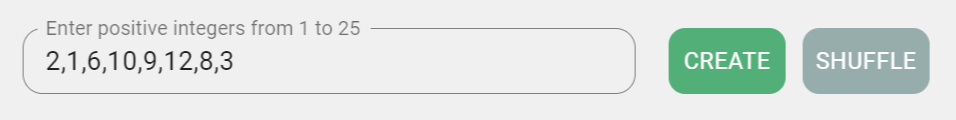
    setCurrentStep(newValue);

    setBars(item);

};

Code 7

**InputBar**

InputBar allows user to type in numbers to create a self-defined array of bars for visualisation. This component has the functionality to restrict user input. An error message will be shown if users tried to visualise an array with a large number or size or characters other than integers. A shuffle button shuffles the bars with random size and numbers.

This is realised by modifying a third-party input component from Material-UI library [2] and a random method from lodash.

// to produce a random array

const shuffle = () => {

    const length = random(3,MAXLENGTH-4);

    let array = [];

    for (let i = 0; i < length; i++) {

        // const element = array[i];

        array

        .push(random(1,MAXNUMBER-5));

    }

    setIsValid(true);

    setWrongMsg(" ");

    setArr(array);

    setStr(array.join(","));

}

Code 8