F21DV Coursework Part 1 Report

Jonathan Song Yang, Lee (H00255553)

Demonstrated to: Amit Parekh (28/01/2022)

Contents

1 Introduction **2**
1.1 General Setup 2
1.2 Individual Exercise HTMLs Setup 5

2 Exercises **6**
2.1 Exercise 1 6
2.2 Exercise 2 7
2.3 Exercise 3 & 4 8
2.4 Exercise 5 9
2.5 Exercise 6 10
2.6 Exercise 7 11
2.7 Exercise 8 12
2.8 Exercise 9 13
2.9 Exercise 10 15
2.10 Exercise 11 17
2.11 Exercise 12 & 13 18
2.12 Exercise 14 & 15 20
2.13 Exercise 16 21
2.14 Exercise 17 22
2.15 Exercise 18 & 19 23
2.16 Exercise 20 25
2.17 Exercise 21 26
2.18 Exercise 22 27
2.19 Exercise 23 31
2.20 Exercise 24 32
2.21 Exercise 25 to 27 33
2.22 Exercise 28 35
2.23 Exercise 29 36
2.24 Exercise 30 & 31 37
2.25 Exercise 32 39

1 Introduction

The entirety of the F21DV four-part coursework is done in a way where it resembels a full static or a `node.js` server-side web application. The goal of this series of coursework is to demonstrate the understanding of `d3.js`.

Github Repo: https://github.com/jonleesy/F21DV-Coursework
Github Pages: https://jonleesy.github.io/F21DV-Coursework/public

1.1 General Setup

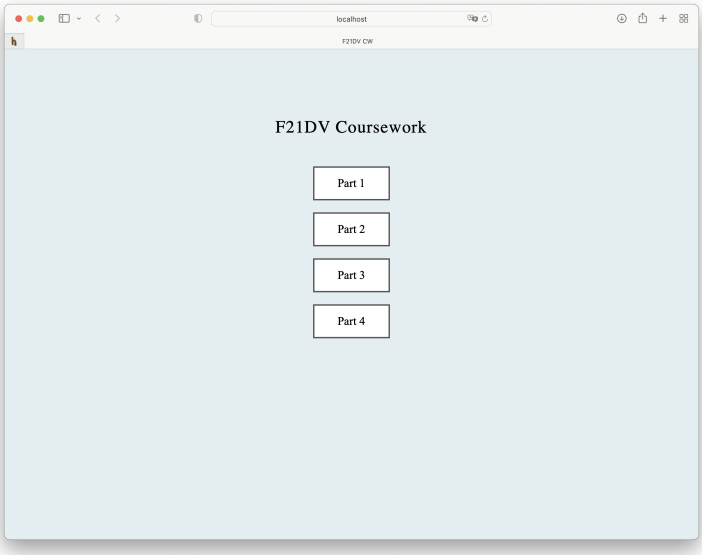

Figure 1.1: Main Index Page

The main index page of the web application would show buttons to access parts 1 to 4, as shown in figure 1.1, and for the case of Lab 1, upon clicking on the Part 1 button, a series of urls linking to different exercises would be shown, as seen in figure 1.1 below.

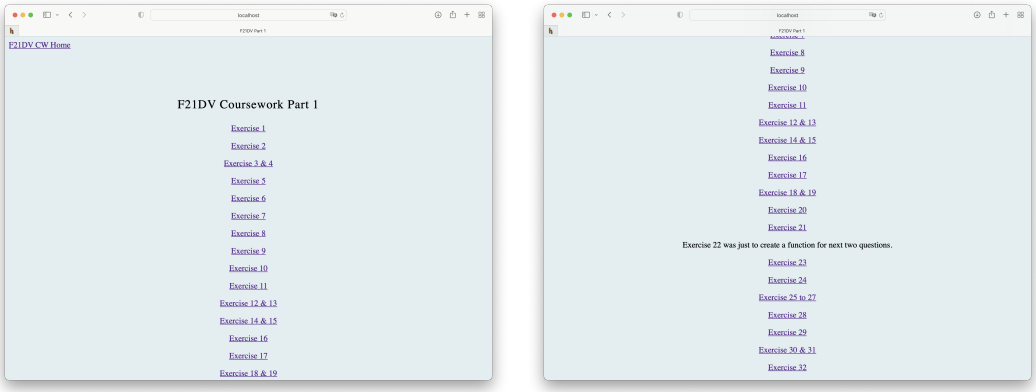

Figure 1.2: Main Index Page

To demonstrate the understanding of `d3.js`, this series of URLs were created systematically using `d3.js`, as shown in listing 1.1 abstact below.

```
1   // Array of numbers [0, 1, 2, ..., 32] defining number of exercises
2   const data = Array.from(Array(33).keys());
3
4   // Creating links to 32 Exercises Systematically.
5   d3.select('body').selectAll('p')
6       .data(data)
7       .enter()
8           // Append a <p> for each exercise and add an <a> for each <p>
9           .append('p').style('text-align', 'center')
10              .attr('class', d => 'task' + d)
11              .append('a')
12                  .attr('href', d => 'task' + d +'.html')
13                  .html(d => 'Exercise ' + d);
```

Listing 1.1: Systematic URL Creation

As the lab exercise would involve certain combinable exercises, such as exercises 30 and 31, I have also created a generalised function to help **combine** and **delete** certain exercises.

```
1   /**
2   * General Merge Function. If its just 2 consecutive exercise, ignore 3rd and 4th
    parameter.
3   * However, if merge is for a range of exercises, spanning more than 2, only enter
     the
4   * first and last exercise number.
5   * @param {*} first first exercise to merge
6   * @param {*} second second exercise to merge. Last exercise if there are more
    than 2 exercises.
7   * @param {*} cond1 used to change exercise <number> to Exercise <number> to <
    number>
8   * @param {*} cond2 used to rename html file to task<first>n<second>.html
9   */
10  function mergeTask(first, second, cond1 = ' \& ', cond2 = 'n') {
11      d3.select('.task' + second).remove();
12      d3.select('.task' + first + ' a').html('Exercise ' + first + cond1 + second)
13                                          .attr('href', d => 'task' + first + cond2
    + second + '.html');
14  }
```

Listing 1.2: Systematic URL Removal

Listing 1.2 shows a function that renames the href of combined exercises, and removes and rename a pre-existing href. For example, we have initially Exercises 30 and 31. The function first removes the

exercise 31's <p>, and then renames the original html href file from task30.html to task30n31.html.

```
1     // Remove task function
2     function removeTask(task, message) {
3         d3.select(`.task${task} a`).remove();
4         d3.select(`.task${task}`).append('p')
5                                  .text(message);
6     }
```

Listing 1.3: Systematic ¡a¿-text Replacement

Listing 1.3 shows a generalised function that replaces the exercises URL with a message string. For example, in exercise 22, we were asked to create a generalised SVG and lines function. This function will now remove the href from the <p> element, and replace it with a message string.

1.2 Individual Exercise HTMLs Setup

Within each exercise's own HTML `body`, they all inherit a generic CSS setting for a div called `.answerCenter` which acts as the center element for the presentation of answers for each exercise. There is also a generic button cssd style called `.buttonOri` and `.button` that handles the CSS transition of all the buttons.

For each exercise, I have also used a generalised function in `functions.js` to create the `<div>`s and buttons so that the styles remain consistent across all exercises.

```
1     /**
2      * Create div's for each question systematically.
3      * @param {*} exerciseNumber Task number.
4      */
5     export function createDiv(exerciseNumber) {
6         d3.select('body')
7             .append('div')
8                 .attr('class', 'container')
9                 .append('div')
10                    .attr('class', 'answerCenter')
11                    .append('p')
12                        .append('strong')
13                            .text('Exercise ' + exerciseNumber + ':')
14    }
15
16    /**
17     * Creates a button to execute action for each task.
18     * @param {*} exerciseNumber
19     */
20    export function createButton(exerciseNumber) {
21        d3.select('body')
22            .append('div').attr('class', 'container')
23                .append('div').attr('class', 'center')
24                    .append('button')
25                        .attr('class', 'buttonori')
26                        .text('Click for Task ' + exerciseNumber)
27    }
```

Listing 1.4: Systematic div and button creation

Listing 1.4 shows the systematic approach used to create the answer container, as well as buttons to execute exercises actions. However, not all exercise would use the button, since not all exercise is in need of an action.

2 Exercises

2.1 Exercise 1

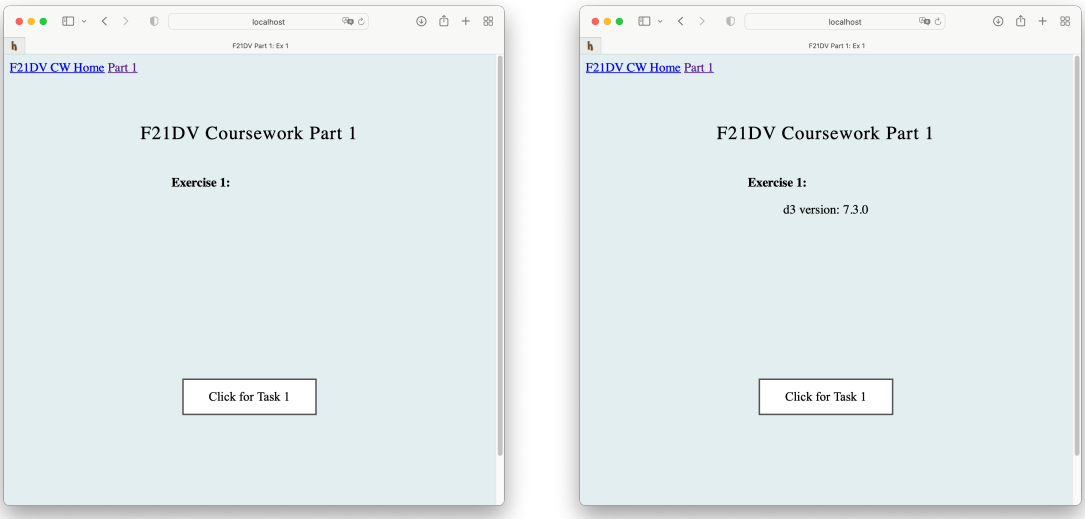

Figure 2.1: Exercise 1

Figure 2.1 shows the outcome after clicking on the action button. The answer container space would print the d3 version number upon clicking the task button.

```
1  // js script for part 1 Exercise:
2  const ex = 1;
3
4  // Create Divs and button systematically using a genral function.
5  import {createDiv, createButton} from '../functions.js';
6  createDiv(ex);
7
8  // Create empty <p> to print version.
9  d3.select('.answerCenter')
10     .append('p')
11         .attr('id', 'task' + ex)
12         .attr('position', 'absolute')
13         .style('text-align', 'center');
14
15 // Button for task action.
16 createButton(ex);
17
18 // Button action: shows d3 version in <p> field.
19 d3.select('.buttonori').on('click', function(){
20     d3.select('#task' + ex).text('d3 version: ' + d3.version);
21 });
```

../../public/js/part1/task1.js

Upon clicking the button, using d3, the button action replaces the empty <p> with "d3 version: 7.3.0". For the remaining exercises, the button action would do a similar thing as well. It would select the desired object for modification, and through a button on-click event listener, modify the selected object.

2.2 Exercise 2

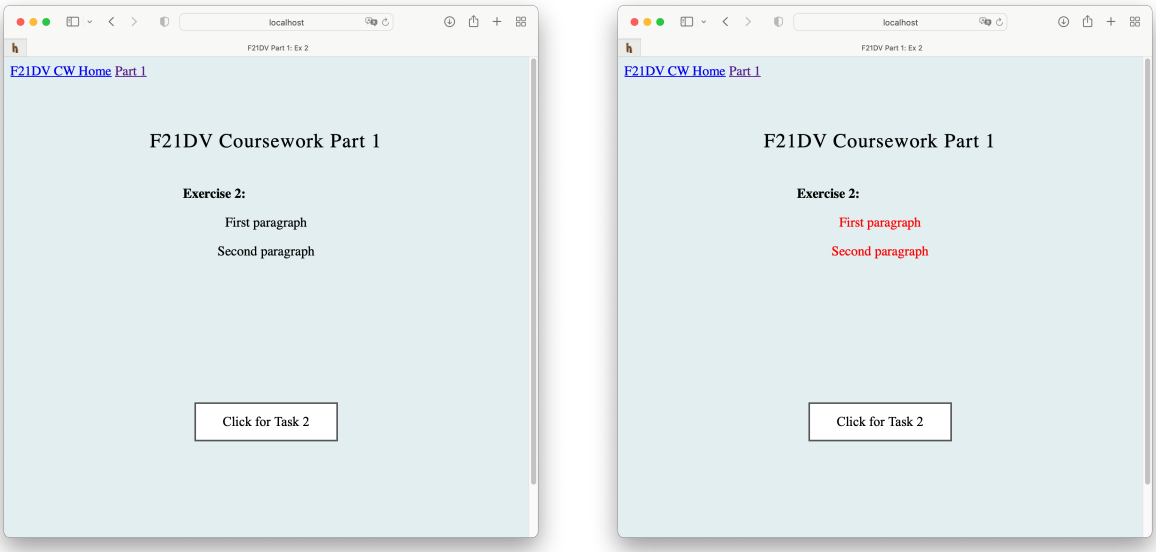

Figure 2.2: Exercise 2

Figure 2.2 shows that upon clicking the button, d3 would change the colours of the selected `<p>`s to red.

2.3 Exercise 3 & 4

Exercises 3 & 4 are one of the exercises where I have combined into one.

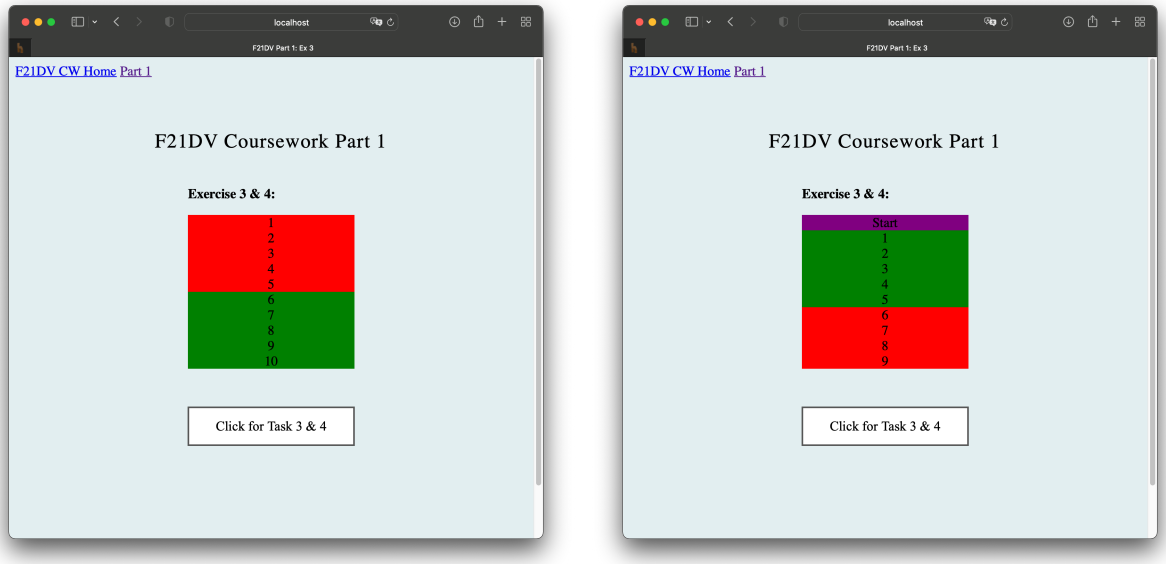

Figure 2.3: Exercise 3

Exercise 3 was to populate 10 div elements with numbers from 1 to 10, and to colour them according to their values. Task four asks then, to replace the first value, “1”, with the text “start”. However, I went ahead and implemented the change for the remaining divs as well, changing their values so that the divs would show start, 1, 2, ..., 9, whilst keeping the colour requirements. Upon clicking the button, the state of the div’s would return to the original state again. Button is clickable multiple times.

2.4 Exercise 5

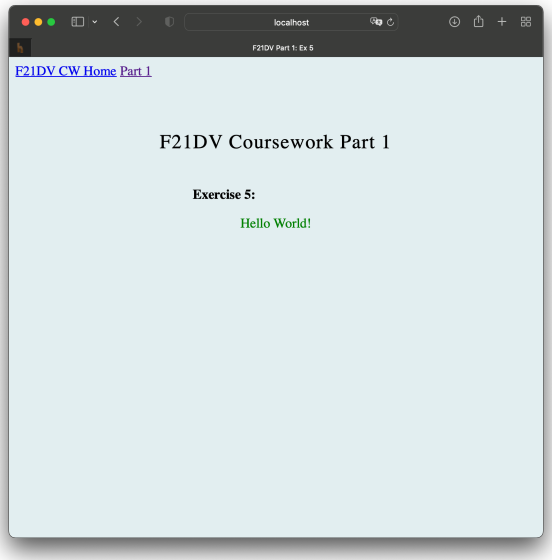

Figure 2.4: Exercise 5

```
1  // js script for part 1 Exercise:
2  const ex = 5;
3
4  // Create Divs and button systematically using a genral function.
5  import {createDiv} from '../functions.js';
6  createDiv(ex);
7
8  // Create <div> for chaining modification.
9  d3.select('.answerCenter')
10     .append('div')
11         .style('text-align', 'center')
12         .style('color', 'green')
13         .text('Hello World!');
```

../../public/js/part1/task5.js

Exercise 5 shows a chained d3 syntax where d3 appends a div, adds a text and changes the text colour to green all in one go.

2.5 Exercise 6

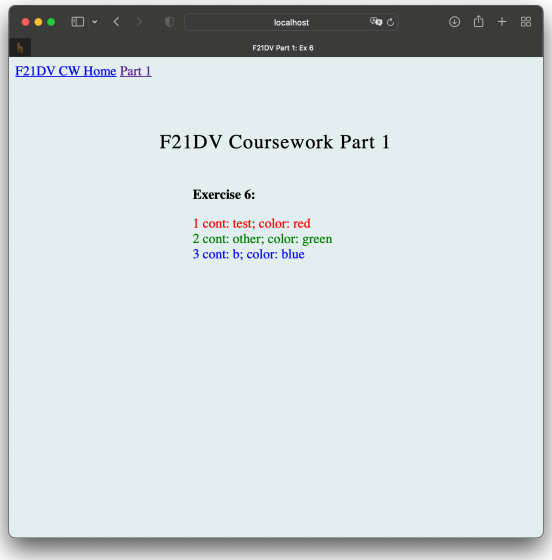

Figure 2.5: Exercise 6

```
1  // js script for part 1 Exercise:
2  const ex = 6;
3
4  // Create Divs and button systematically using a genral function.
5  import {createDiv} from '../functions.js';
6  createDiv(ex);
7
8  // Data for <div>s' color.>
9  const data = [{name:'test', val:1, color:'red'},
10               {name:'other', val:2, color:'green'},
11               {name:'b',     val:3, color:'blue'}];
12
13 // Text + style the <div>s.
14 // Join - Enter was used to create new <div>s.
15 d3.select('.answerCenter')
16     .selectAll('div')
17     .data(data)
18     .join(enter => enter.append('div'))
19     .text((d, i) => `${i + 1} cont: ${d.name}; color: ${d.color}`)
20     .style('color', (d, _) => d.color);
```

../../public/js/part1/task6.js

Exercise 6 adds an additional variable `color`, and prints that out in the text field, on top of what was originally there. I went an extra mile to style each div according to its `color` value.

2.6 Exercise 7

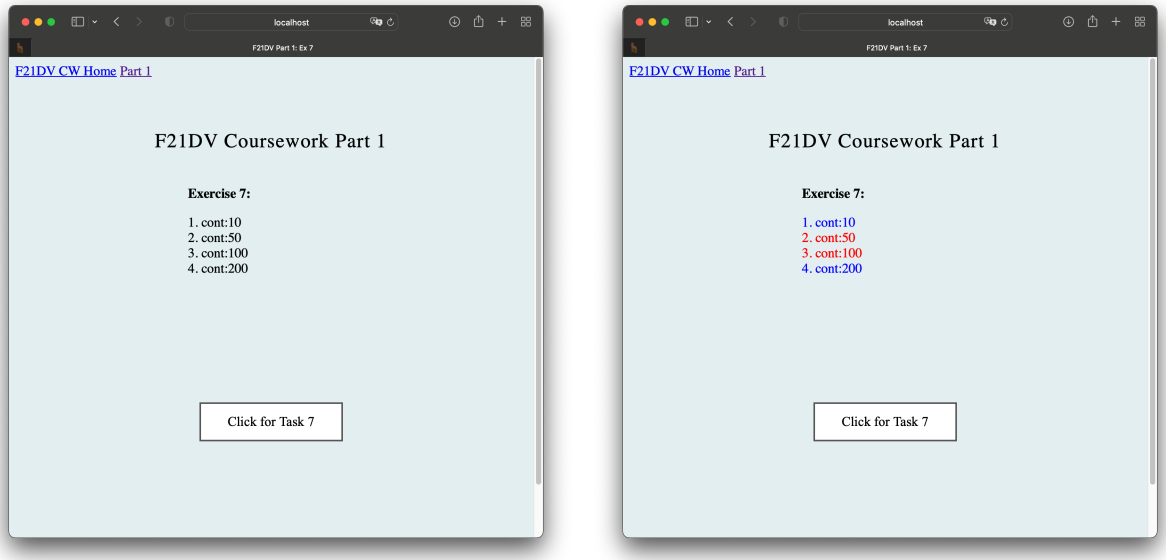

Figure 2.6: Exercise 7

Exercise 7 would change the colour of the div based on the values of the div. Upon click, the button would change the div's with value of numbers between 50 and 100 to red, and the remaining ones to blue. Upon clicking again, the button would revert the state of the divs to their original ones.

2.7 Exercise 8

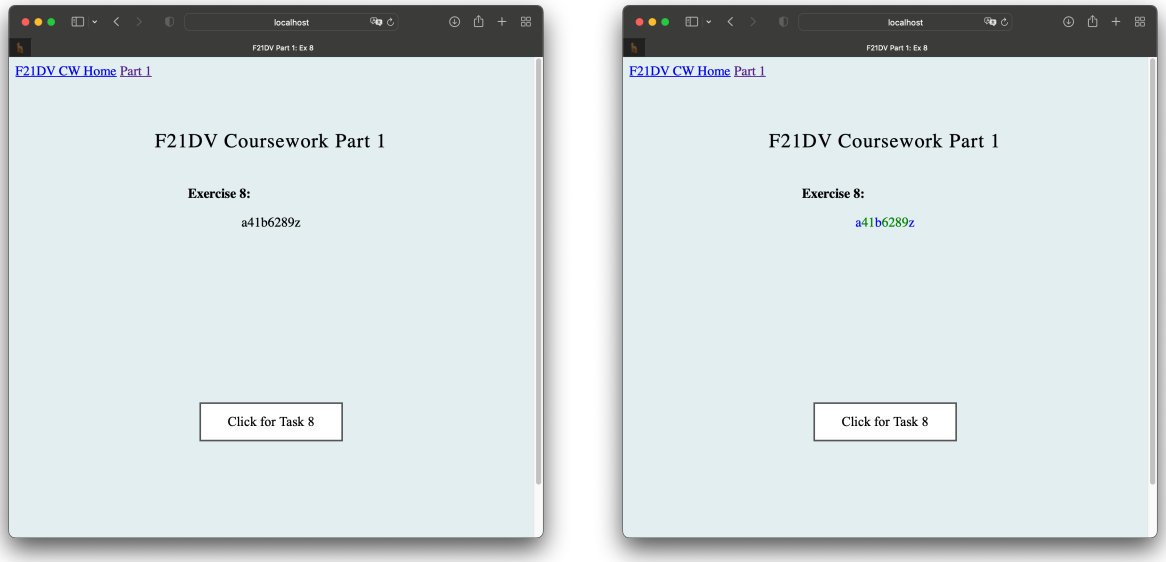

Figure 2.7: Exercise 8

Exercise 8's goal is to set the text colour of a string made out of the `span`s, where if a character is a number, its colour is green, and if its an alphabet, its colour is blue.

2.8 Exercise 9

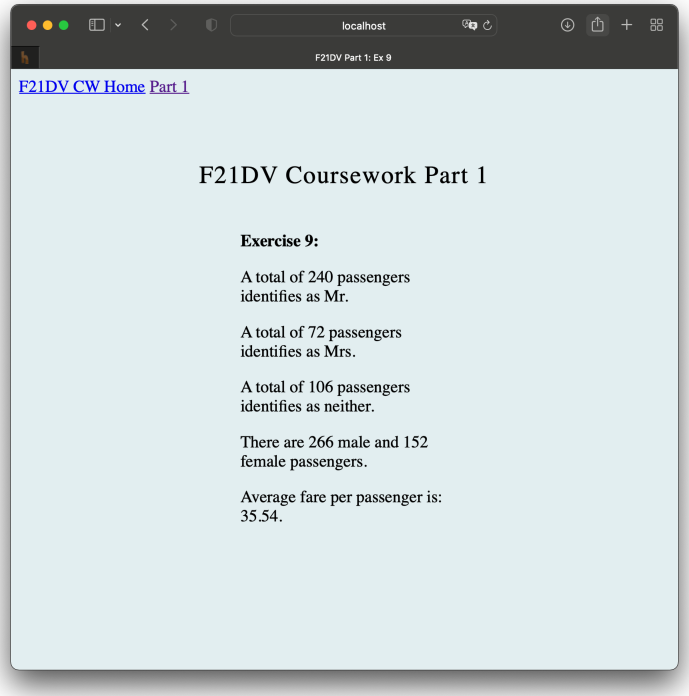

Figure 2.8: Exercise 9

```
1  // js script for part 1 Exercise:
2  const ex = 9;
3
4  // Create Divs and button systematically using a genral function.
5  import {createDiv} from '../functions.js';
6  createDiv(ex);
7
8  // Data from csv.
9  let dataCsv = 'https://raw.githubusercontent.com/dsindy/kaggle-titanic/master/data/
       test.csv';
10
11 // Using a predefined variable for d3 in queston 9.
12 let selection9 = d3.select('.container .answerCenter');
13 d3.csv(dataCsv).then(function(data) {
14     // Local Variables for <p> print statements.
15     let mr = 0, mrs = 0, other = 0;
16     let male = 0, female = 0;
17     let totalFare = 0;
18
19     // Iterate thrugh every entry of data, asynchroneous method.
20     data.forEach(function(d) {
21         // Using local variables to store names and sex for if statements.
22         let n = d.Name;
23         let s = d.Sex;
24
25         // Add 1 for each entries that have 'Mr.' and 'Mrs.' respectively.
26         if (n.includes('Mr.')) {
27             mr ++;
28         } else if (n.includes('Mrs.')) {
29             mrs ++;
30         } else {
31             other ++;
32         }
```

```
33
34          // To check and count entries to see how many male and female on board.
35          if (s === 'male') {
36              male ++;
37          } else if (s == 'female') {
38              female ++;
39          }
40
41          // Calculate the average fare for passenger.
42          if (!Number.isNaN(parseFloat(d.Fare))) {
43              totalFare += parseFloat(d.Fare);
44          }
45      });
46
47      // Using results to append paragrpahs for this div.
48      selection9.append('p').text('A total of ' + mr.toString() + ' passengers
        identifies as Mr.');
49      selection9.append('p').text('A total of ' + mrs.toString() + ' passengers
        identifies as Mrs.');
50      selection9.append('p').text('A total of ' + other.toString() + ' passengers
        identifies as neither.');
51      selection9.append('p').text('There are ' + male + ' male and ' + female + '
        female passengers.');
52      selection9.append('p').text('Average fare per passenger is: ' + Number((totalFare
        /(data.length)).toFixed(2)) + '.');
53  });
```

../../public/js/part1/task9.js

Exercise 9 reads in the csv data, then processes it to count the number of passengers with a Mr. or Mrs. title, and also counts the number of male/female passengers. I have also added on a calculation that calculates the average fare price per titanic passenger. This was done by adding all non-NaNs to a local variable, then dividing it by total passenger.

2.9 Exercise 10

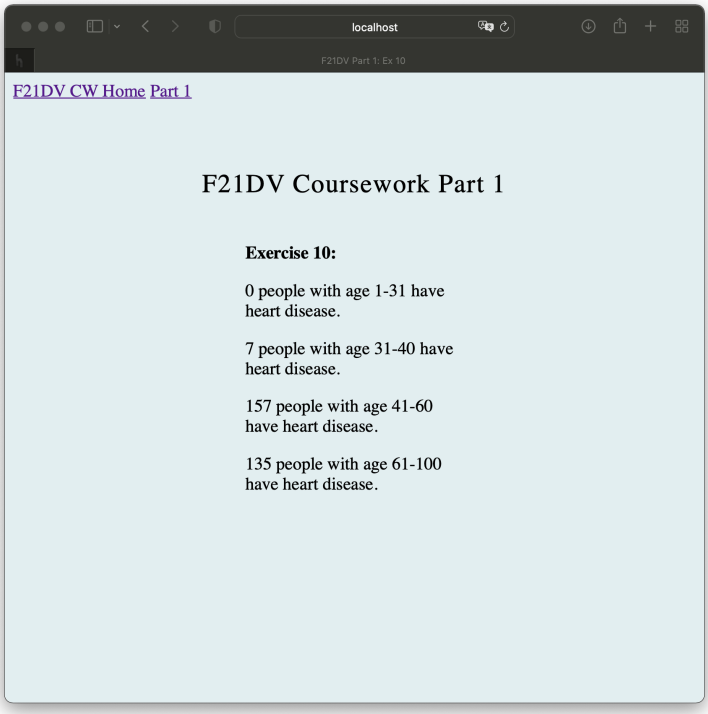

Figure 2.9: Exercise 10

```
1  // js script for part 1 Exercise:
2  const ex = 10;
3
4  // Create Divs and button systematically using a genral function.
5  import {createDiv} from '../functions.js';
6  createDiv(ex);
7
8  // Data from csv
9  let dataCsv = d3.csv('https://raw.githubusercontent.com/akmand/datasets/master/
       heart_failure.csv');
10
11 // Function that gets the numbers for this task and task 14 & 15
12 async function getData() {
13     const data = await dataCsv;
14     let young = 0;
15     let mid = 0;
16     let old = 0;
17     let veryold = 0;
18     for (const row of data) {
19         if (parseInt(row.age) <= 30) {
20             young ++;
21         } else if (parseInt(row.age) <= 40) {
22             mid ++;
23         } else if (parseInt(row.age) <= 60) {
24             old ++;
25         } else {
26             veryold ++;
27         }
28     }
29     // Adding a '0' since d3 will skip "young = 0".
30     return ['0', young, mid, old, veryold]
31 }
32
```

```
33 // Bind above variables to be used as data.
34 d3.select('.answerCenter').selectAll('p')
35     .data(await getData())
36     .enter()
37         .append('p')
38         .text(function(d, i) {
39             let msg = d.toString() + ' people with age ';
40             switch (i) {
41                 case 1: msg += '1-31'; break;
42                 case 2: msg += '31-40'; break;
43                 case 3: msg += '41-60'; break;
44                 case 4: msg += '61-100';
45             }
46             msg = msg + ' have heart disease.';
47             return msg;
48         });
49
50 // Function for task 14 & 15.
51 export async function exportData() {
52     const expData = await getData();
53     return [{name: "1-31", value: expData[1]},
54             {name: "31-40", value: expData[2]},
55             {name: "41-60", value: expData[3]},
56             {name: "61-100", value: expData[4]},]
57 }
```

../../public/js/part1/task10.js

Exercise 10 is pretty much the same as exercise 9. Calling the right variable `name` and then processing its values. However, since the heart data is being used again at a later exercise, I have created an async function that exports the data, so that the process is only done once.

2.10 Exercise 11

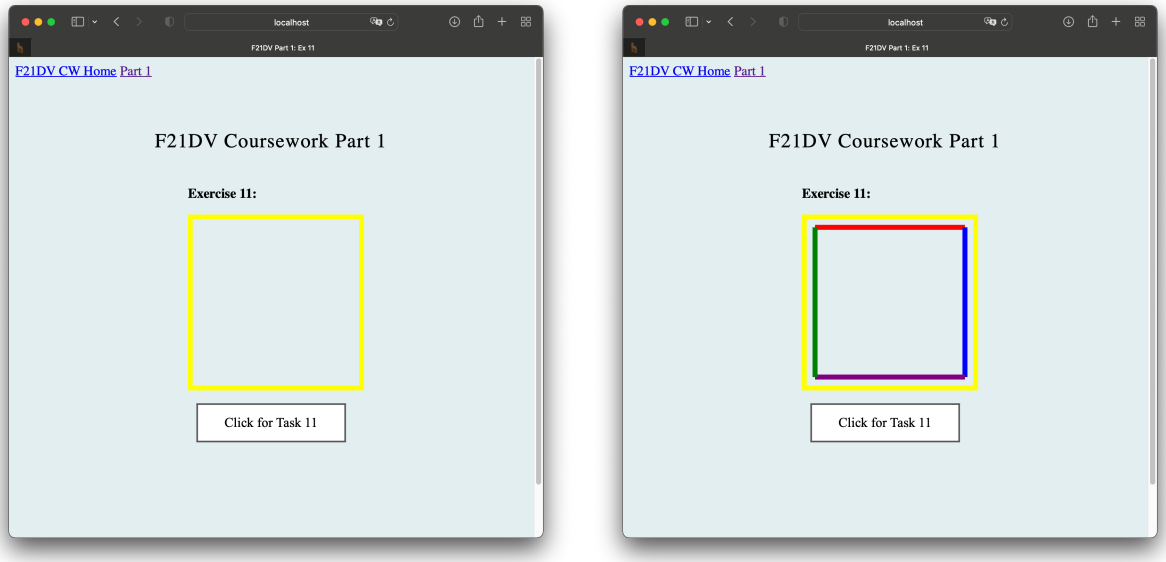

Figure 2.10: Exercise 11

Exercise 11, we first append a square yellow svg, and on the click of the button, 4 more lines are plot in the middle of the svg.

2.11 Exercise 12 & 13

name	width	height	x1	y1	x2	y2	x	y	cx	cy	r	rx	ry	stroke	stroke-width	text	fill
rect	100	50					50	0									pink
circle									100	75	25						green
ellipse									100	125		75	25				blue
text							75	175						black		sometext	red
line			10	10	190	190								purple	6		

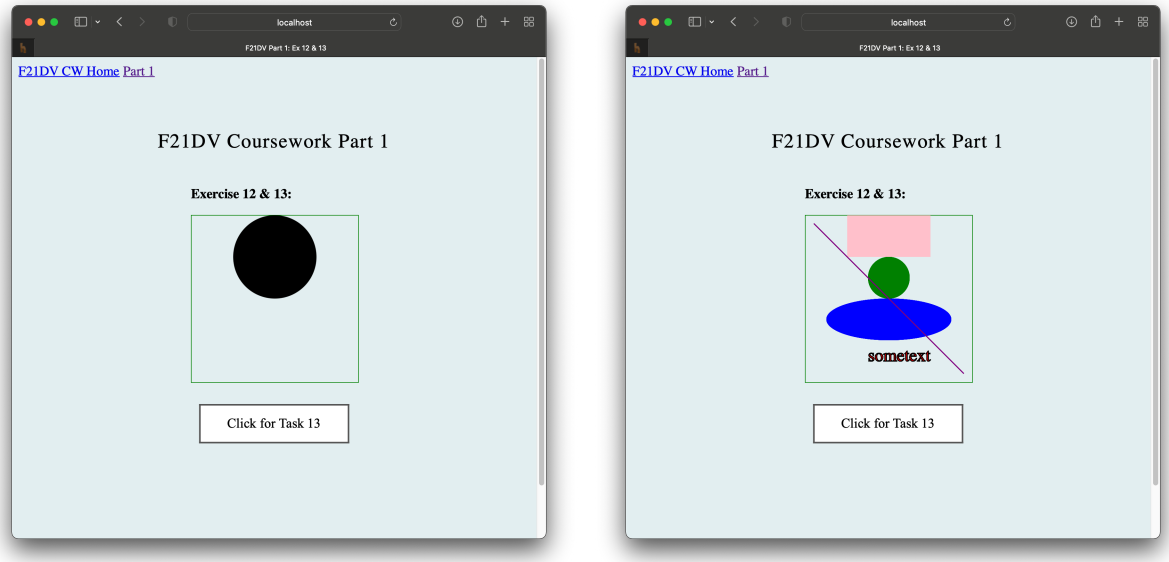

Figure 2.11: Exercise 12 & 13

```
// js script for part 1 Exercise:
const ex = '12 \& 13';

// Create Divs and button systematically using a genral function.
import {createDiv, createButton} from '../functions.js';
createDiv(ex);

// Read Csv, as part of exercise 12.
const data12 = d3.csv('../../data/part1/task12.csv');

// Adding an SVG element.
d3.select('.answerCenter')
    .append('svg')
        .attr('width',  200)
        .attr('height', 200)
        .style('border', '1px solid green')
        .append('circle')
            .attr('cx', 100)
            .attr('cy', 50)
            .attr('r', 50);

// Create action button.
createButton(13)

// Button action: add new svg elements and update-remove existing ones.
d3.select('.buttonori').on('click', function(){
    data12.then(function(data) {
        data.forEach(function(d) {
            let name = d.name;
            let data = new Array();
            let text = d.text;

            // Filtering for non-empty attrs.
            for (const col in d) {
```

```
35                  if (d[col] !== '' && col !== 'text') {
36                      data.push({key: col, val: d[col]});
37                  }
38              }
39
40              // Add dummy attr for shorter shapes.
41              while (data.length < 6) {
42                  data.push({key: 'dummy', val: ''})
43              }
44
45              // Enter-Update-Exit demo-ed for question 13.
46              d3.select('svg')
47                  .selectAll(name)
48                  .data(data)
49                  .join(
50                      enter => enter.append(data[0].val)
51                                  .attr(data[1].key, data[1].val)
52                                  .attr(data[2].key, data[2].val)
53                                  .attr(data[3].key, data[3].val)
54                                  .attr(data[4].key, data[4].val)
55                                  .attr(data[5].key, data[5].val)
56                                  .text(text),
57                      exit => exit.transition()
58                                  .duration(2000)
59                                  .attr('r', 0)
60                                  .remove()
61                  )
62          });
63      })
64  });
```

../../public/js/part1/task12n13.js

This exercise starts off with reading the CSV data, as shown on top of figure 2.11. For each shape to be plot on, we enter its relevant attribute values and leave the irrelevant ones empty.

Next, upon the click of the button, there would first be a filtering of the data, where only relevant (non-empty) attributes key-value pair is kept, in temporary local variable — `data`. For shapes with lesser attributes, we append more dummy data, just so that the number of attributes are all the same, allowing us to add shapes systematically, and to a point of not needing to have a few callback functions. I have also added an exit transition for the initial 'circle' shape, just to demonstrate the use of a join function.

2.12 Exercise 14 & 15

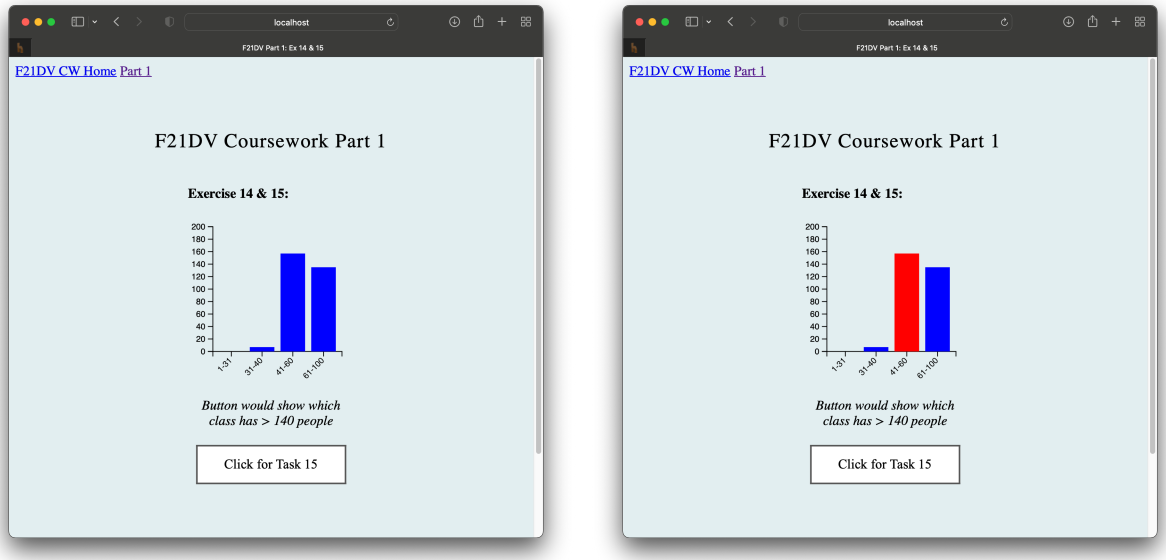

Figure 2.12: Exercise 14 & 15

Exercise 14 & 15 are the exercises using the exported async data from question 10. The question asked only for rectangles, but I have added the axes on top of it. Hence, importing the data, we now can categorise them according to value and scale them accordingly to determine the rectangle height in this bar chart.

Upon click, the button would trigger an update action for the rectangle `style` colours. If the number of people with heart disease for said age group is more than 140, the colour of said rectangle would be turned red.

2.13 Exercise 16

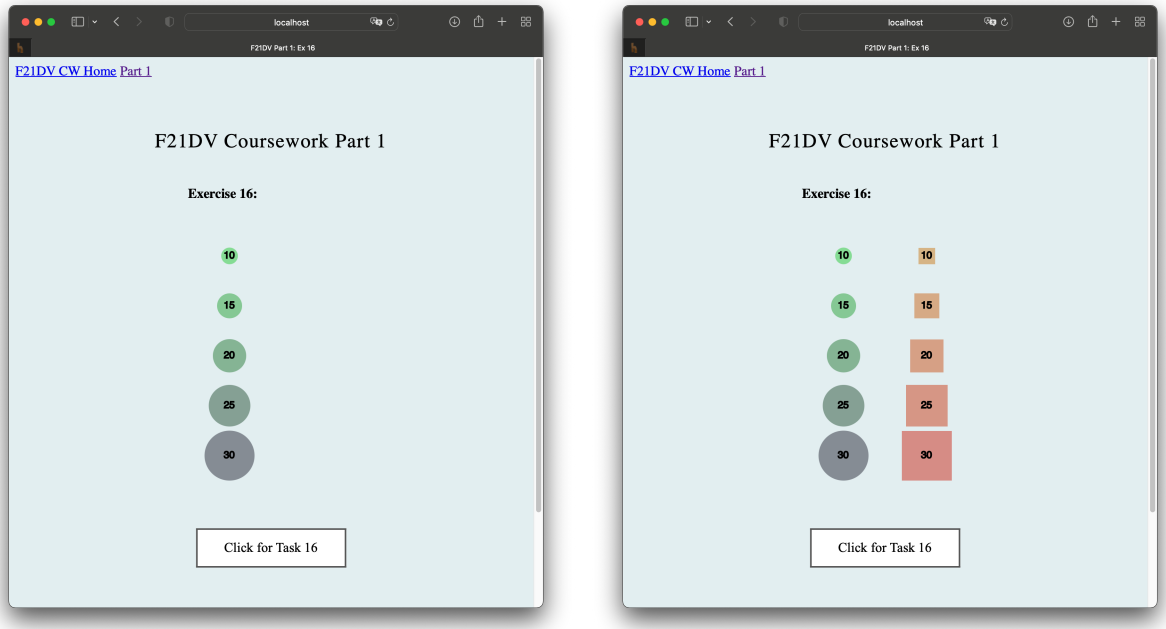

Figure 2.13: Exercise 16

Exercise 16 is about adding shapes based on the given data and its index within the data array. The example given shows how to add circles following the data values, which determines its size. The exercise also uses index numbers to determine its vertical axis translation.

By pressing the button, we would trigger the squares to appear. I have done the same for the square, just instead of using the data values to set the length of the sides of square, I multiply it by 2, just so that the height of the square would be the same as the diameter of the circle. The verticle translation is also based off the index of data points, but the difference is that there is an extra few values in the horizontal translation of the squares, so that it can now be side by side to the circles.

2.14 Exercise 17

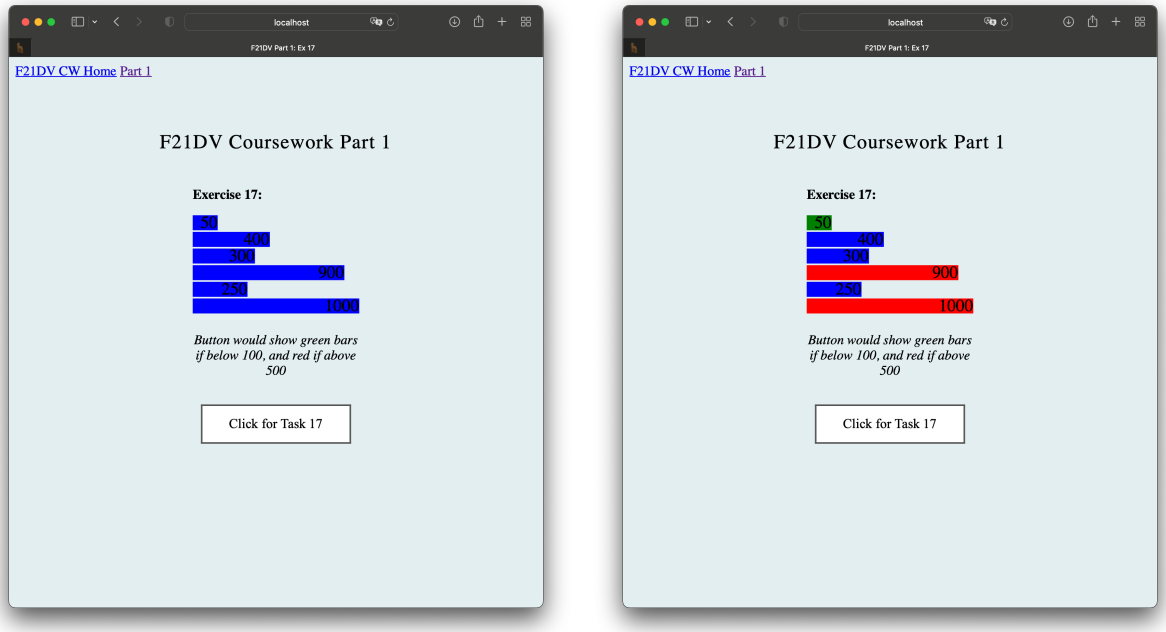

Figure 2.14: Exercise 17

Exercise 17 is just a simple horizontal bar chart, like the example from Exercise 14 and 15. We first create an svg, and then slowly add rectangles onto it, and aligning it using transform-translations. The button then changes the colour of the rectangles depending on the value.

2.15 Exercise 18 & 19

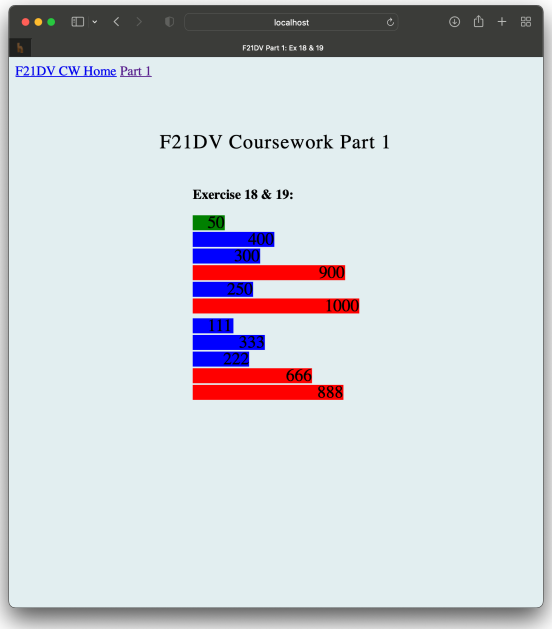

Figure 2.15: Exercise 18 & 19

```
1  // js script for part 1 Exercise:
2  const ex = '18 \& 19';
3
4  // Create Divs and button systematically using a general function.
5  import {createDiv, createButton} from '../functions.js';
6  createDiv(ex);
7
8  // Data for bar chart.
9  const data1 = await d3.csv('../../data/part1/task18.csv');
10 const data2 = await d3.csv('../../data/part1/task19.csv');
11
12 // Svg constants
13 const width = 200, barHeight = 20, margin = 2;
14
15 // Task 18 function.
16 async function task18(data, csvid) {
17
18     // Creating the svg object.
19     const svg = d3.select('.answerCenter')
20                     .append('svg')
21                         .attr('width', width)
22                         .attr('height', barHeight * data.length)
23                         .attr('id', `task${csvid}`);
24
25     // Define the x-axis.
26     const xScale = d3.scaleLinear()
27                         .domain([0, 1000])
28                         .range([30, 200]);
29
30     // Add 'g' object.
31     const g = svg.selectAll('g')
32                 .data(data)
33                 .enter()
34                     .append('g')
35                         .attr('transform', (_, i) => `translate(0, ${i * barHeight})
    `);
```

```
36
37      // Add 'rect' elements.
38      g.append('rect')
39          .attr('width', d => xScale(d.values))
40          .attr('height', barHeight - margin)
41          .attr('fill', d => (d.values < 100) ? 'green' : ((d.values > 500) ? 'red' : '
        blue'))
42
43      // Add 'text' objects.
44      g.append('text')
45          .attr('x', d => xScale(d.values))
46          .attr('y', barHeight/2)
47          .attr('dy', '.25em')
48          .style('text-anchor', 'end')
49          .text(d => d.values);
50  }
51
52  // Calling the same function for 18, and 19.
53  task18(data1, 1)
54  task18(data2, 2)
```

../../public/js/part1/task18n19.js

Exercise 18 is plotting some bar charts based off some csv data, and question 19 extends that by adding them all into a function so that the function could be called as many times as desired, and with different data each time. The function would append a new SVG to the `.answerCenter` div each time the function is called, and then rectangles would be added onto it afterwards.

2.16 Exercise 20

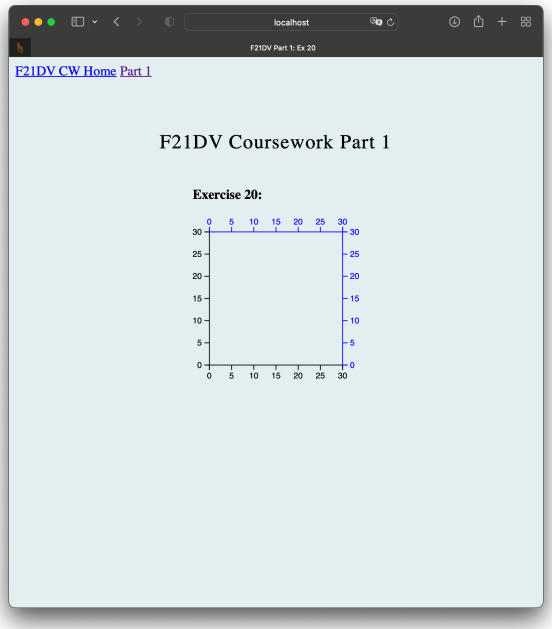

Figure 2.16: Exercise 20

Exercise 20 is just the same as plotting the axes in exercise 14 and 15. The only difference is the two more top and right axis that needs to be plot. This would just have a different `.style()` colour, and also differnt transform-translation values.

2.17 Exercise 21

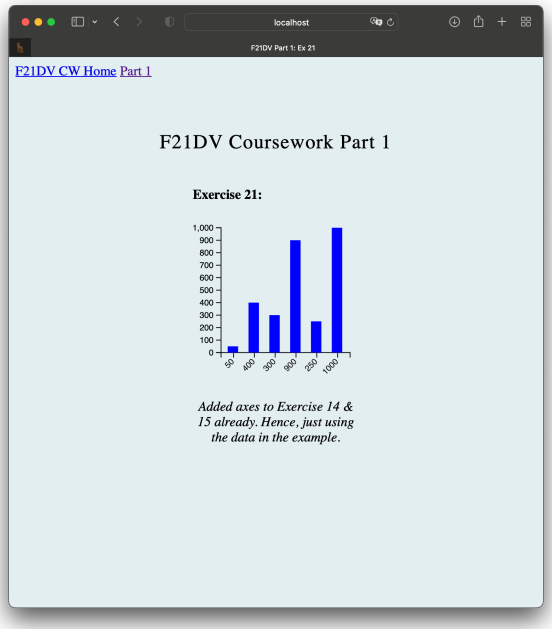

Figure 2.17: Exercise 21

What was requested in exercise 21 was done in exercise 14 and 15, hence just using different data values from the example given for this.

2.18 Exercise 22

```
// js script for part 1 Exercise:
const ex = 22;

// Create Divs and button systematically using a general function.
import {createDiv} from '../functions.js';;

/**
 * Creates and appends the SVG and its axes.
 * @param {*} data data of plots
 * @returns Svg element, the X and Y axis scaler.
 */
export function createSvg(data, task) {

    // Setting up div
    createDiv(task)

    // Svg dimension constants.
    const xSize = 400, ySize = 400, margin = 40, xMax = xSize - margin*2, yMax =
    ySize - margin*2;

    // Increasing the size of the answer container
    d3.select('.container').style('height', `${ySize}px`);
    d3.select('.answerCenter').style('width', `${xSize}px`);

    // limits of data.
    const xExtent = d3.extent(data, d => d.x);
    const yExtent = d3.extent(data, d => d.y);

    // Create and apend the svg object.
    const svg = d3.select('.answerCenter')
                  .append('svg')
                      .attr('width', xSize)
                      .attr('height', ySize)
                      .append('g')
                          .attr('transform', `translate(${margin}, ${margin})`);

    // Define the horizontal scale.
    const horScale = d3.scaleLinear()
                      .domain([xExtent[0], xExtent[1]])
                      .range([0, xMax]);

    // Define the vertical scale.
    const verScale = d3.scaleLinear()
                      .domain([yExtent[0], yExtent[1]])
                      .range([yMax, 0]);

    // Add Axis: https://www.d3-graph-gallery.com/graph/custom_axis.html
    // x axis.
    svg.append('g')
        .attr('transform', `translate(0, ${yMax})`)
        .call(d3.axisBottom(horScale));

    // y axis.
    svg.append('g')
        .call(d3.axisLeft(verScale));

    // Top axis.
    svg.append('g')
        .call(d3.axisTop(horScale))
        .style('color', 'blue');

    // Right axis.
    svg.append('g')
        .attr('transform', `translate(${xMax}, 0)`)
```

```
64          .call(d3.axisRight(verScale))
65          .style('color', 'blue');
66
67      return {svg, horScale, verScale};
68  }
69
70  /**
71   * Function that adds a line to the SVG object.
72   * @param {*} data Same data as createSvg().
73   * @param {*} svgObj the created SVG Object.
74   * @param {*} color colour for each line.
75   */
76  export function addLines(data, svg, color) {
77      // Create and adds the lines
78      svg.svg
79          .append('path')
80              .datum(data)
81              .attr('fill', 'none')
82              .attr('stroke', color)
83              .attr('stroke-width', 2)
84              .attr('d', d3.line()
85                      .x(d => svg.horScale(d.x) )
86                      .y(d => svg.verScale(d.y))
87                      )
88  }
89
90  /**
91   * Function adds lines with datapoints on the plot.
92   * @param {*} data x, y coordinates data.
93   * @param {*} svg svg element.
94   * @param {*} color coloour of line and dots.
95   * @param {*} shape shape of points.
96   */
97  export function addLinesShape(data, svg, color, shape) {
98      // Add lines first. This function works independently from addLines().
99      addLines(data, svg, color);
100
101     // Pre-defining the datapoint symbol.
102     let symbol;
103     switch (shape) {
104         case 'triangle': symbol = d3.symbol().type(d3.symbolTriangle).size(20); break
        ;
105         case 'circle': symbol = d3.symbol().type(d3.symbolCircle).size(20); break;
106     }
107
108     // Add data points.
109     const dots = svg.svg.selectAll('.dots')
110                         .data(data)
111                         .enter()
112                             .append('path')
113
114     dots.attr('d', symbol)
115         .attr('fill', color)
116         .attr('stroke', 'black')
117         .attr('id', `${shape}`)
118         .attr('stroke-width', 0.5)
119         .attr('transform', function(d) {
120             return `translate(${svg.horScale(d.x)}, ${svg.verScale(d.y)})`
121         })
122 }
123
124 /**
125  * Function adds coordinates of desired datapoints on the plot.
126  * @param {*} data x, y coordinates data.
127  * @param {*} svg svg element.
128  * @param {*} pointNumber which point would you like to plot.
```

```
 */
export function addLinesCoor(data, svg, pointNumber) {
    const margin = 7;
    svg.svg
        .append('g')
        .selectAll('text')
        .data(data)
        .enter()
            .append('text')
                .style('font-size', '12px')
                .attr('transform', d => `translate(${svg.horScale(d.x) + margin},
                                                   ${svg.verScale(d.y) + margin})`)
                .text((d, i) => (i == pointNumber) ? `(${d.x}, ${d.y})` : '')
}

/**
 * Function adds lines with datapoints on the plot.
 * @param {*} data x, y coordinates data.
 * @param {*} svg svg element.
 * @param {*} color coloour of line and dots.
 * @param {*} shape shape of points.
 */
 export function addLinesShapeDifferentColor(data, svg, interpolateMethod, shape) {
    // Color Scale.
    const colScale = d3.scaleSequential()
                    .domain(d3.extent(data, d => d.y))
                    .interpolator(interpolateMethod);

    // Add lines first. Different colour for different line.
    addLines(data, svg, d => (shape === 'circle' ? 'blue' : 'red'));

    // Pre-defining the datapoint symbol.
    let symbol;
    switch (shape) {
        case 'triangle': symbol = d3.symbol().type(d3.symbolTriangle).size(30); break
    ;
        case 'circle': symbol = d3.symbol().type(d3.symbolCircle).size(30); break;
    }

    // Add data points.
    const dots = svg.svg.selectAll('.dots')
                        .data(data)
                        .enter()
                            .append('path')

    dots.attr('d', symbol)
        // Using the color scale to set the fill colour of points.
        .attr('fill', d => colScale(d.y))
        .attr('stroke', 'black')
        .attr('id', `${shape}`)
        .attr('stroke-width', 0.5)
        .attr('transform', function(d) {
            return `translate(${svg.horScale(d.x)}, ${svg.verScale(d.y)})`
        })
}
```

../../public/js/part1/task22.js

Task 22 has no output as its sole purpose is to provide generalised functions for the remaining questions. Its main functions includes:

- `createSvg()` — which creates the svg element, and returns the svg itself, horizontal scale, and verticle scale.
- `addLines()` — which takes the createSvg() from before, and uses either its svg, hor/vertical scale, and processes some lines and adds to it.

- `addLinesShape()` — same as before, just this time having and extra function that adds data points based on the lines to add.
- `addLinesCoor()` — extends `addLinesShape()` and adds coordinates near the data points along the lines. Users can self identify which points to add.
- `addLinesShapeDifferentColor()` — adds lines and data points (of different shape), just that this time, the colour of data points is dependant on the data value.

2.19 Exercise 23

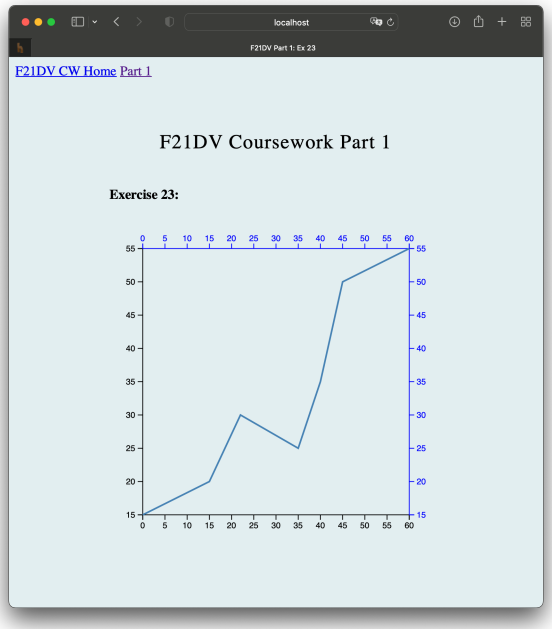

Figure 2.18: Exercise 23

```
1  // js script for part 1 Exercise:
2  const ex = 23;
3
4  // Create Divs and button systematically using a general function.
5  import { createSvg, addLines } from '../task22.js';
6
7  // Import data for task 23.
8  const data = await d3.csv('../../data/part1/task23.csv');
9
10 // Append the svg element
11 const svg = createSvg(data, ex);
12 addLines(data, svg, 'steelblue');
```

../../public/js/part1/task23.js

Exercise 23 reads data containing x, y coordinates from a csv file, and plot these on the svg element using the `createSVG()` and `addLines()` function imported from `task22.js`.

2.20 Exercise 24

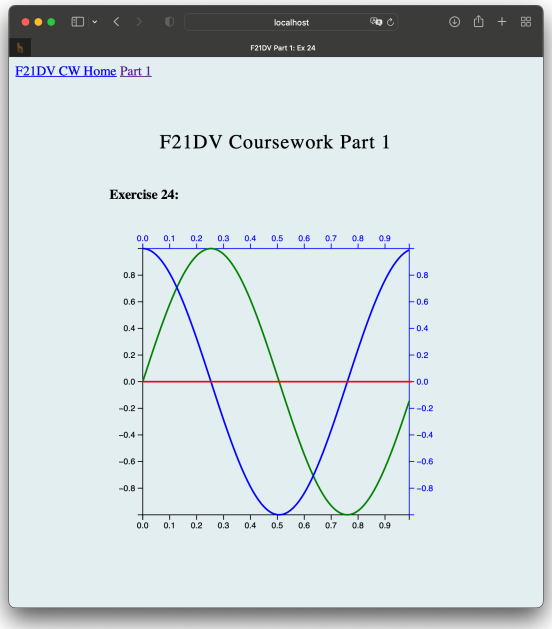

Figure 2.19: Exercise 24

```
1  // js script for part 1 Exercise:
2  const ex = 24;
3
4  // Create Divs and button systematically using a general function.
5  import { createSvg, addLines } from '../task22.js';
6
7  // Creating the sine and cosine data points.
8  const n = 100;
9  const sine = new Array(), cosine = new Array();
10 for (let i = 0; i < n; i++) {
11     sine.push({x: i/100, y: Math.sin(6.2 * i/100)});
12     cosine.push({x: i/100, y: Math.cos(6.2 * i/100)});
13 }
14
15 const middleLine = [{x:0, y:0}, {x:1, y:0}]
16
17 // Append the svg element
18 const svg = createSvg(sine, ex);
19 addLines(sine, svg, 'green');
20 addLines(cosine, svg, 'blue');
21 addLines(middleLine, svg, 'red');
```

../../public/js/part1/task24.js

Same as exercise 22. The difference is just with the self generated sine and cos data points. Once generated, the sine and cosine lines are added, and I added a X axis red line just to show the x axis.

2.21 Exercise 25 to 27

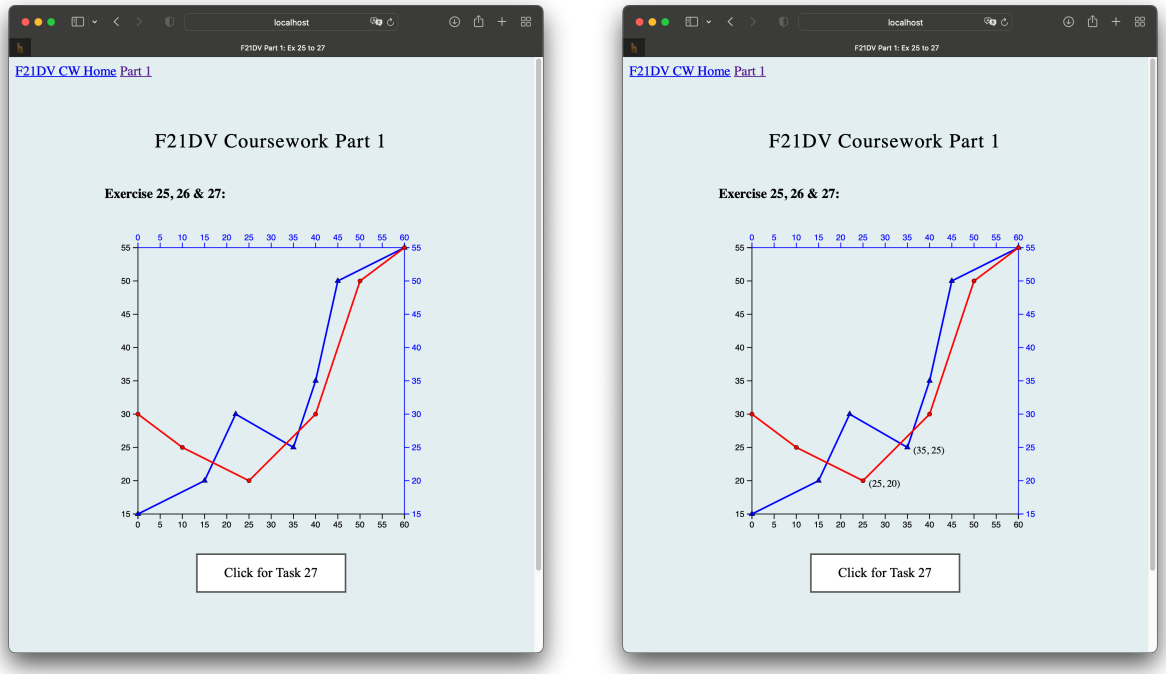

Figure 2.20: Exercise 25 to 27

```
1  // js script for part 1 Exercise:
2  const ex = '25, 26 \& 27';
3
4  // Import create button function.
5  import { createButton } from '../functions.js';
6
7  // Import svg and line function from task22.js.
8  import { createSvg, addLinesShape, addLinesCoor } from './task22.js';
9
10 // Reusing data from task 23.
11 const data = await d3.csv('../../data/part1/task23.csv');
12 // Task 26 data.
13 const data2 = await d3.csv('../../data/part1/task26.csv');
14
15 // Append the svg element
16 const svg = createSvg(data, ex);
17
18 // Line 1.
19 addLinesShape(data, svg, 'blue', 'triangle');
20
21 // Line 2.
22 addLinesShape(data2, svg, 'red', 'circle');
23
24 // Modify container height.
25 d3.select('.container').style('height', '430px');
26
27 // Button for task action.
28 createButton(27);
29
30 // Button active. Button no longer does stuff once its clicked.
31 const buttonActive = true;
32
33 // Button action: adds task 27.
34 d3.select('.buttonori').on('click', function(){
35     while (buttonActive) {
```

```
36          // Add Coor Text
37          addLinesCoor(data, svg, 3);
38          addLinesCoor(data2, svg, 2);
39          buttonActive = false;
40      }
41  });
```

../../public/js/part1/task25to27.js

I have decided to combine exercises 25 and 26, since they are essentially the same thing just with a different shape. Making use of the function `addLinesShape()` from `task22.js`, we simply call the function twice.

For the task in exercise 27 however, I have decided to use a button to complete this. Uppon the press of the button, we add a point to each graph using the `addLinesCoor()` function, and we define the index of the data point we would like to plot. For instance, I plot the 3rd index for the blue line, and the 2nd index for the red line.

2.22 Exercise 28

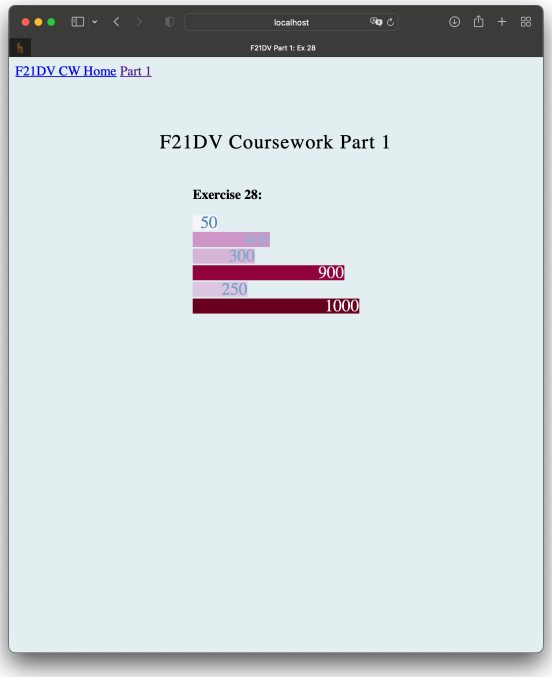

Figure 2.21: Exercise 28

```
1     ...
2     // Colour Scale.
3     const boxCol = d3.scaleSequential()
4                          .domain(d3.extent(data))
5                          .interpolator(d3.interpolatePuRd);
6     const textCol = d3.interpolateRgb('steelblue', 'white');
7
8     // Add 'rect' elements.
9     g.append('rect')
10        .attr('width', d => xScale(d))
11        .attr('height', barHeight - margin)
12        .attr('fill', d => boxCol(d));
13
14    // Add text object to rectangle.
15    g.append('text')
16        .attr('x', d => xScale(d))
17        .attr('y', barHeight/2)
18        .attr('dy', '.25em')
19        .style('text-anchor', 'end')
20        .text(d => d)
21        .style('fill', d => textCol(d/1000));
22    ...
```

Listing 2.1: abstract from task28.js

Exercise 28 is the same as what that was done in exercise 17. The only difference is with the definition of the colour schemes, as shown in listing 2.1. Then using the scale, we obtain the colours for each rectangle.

2.23 Exercise 29

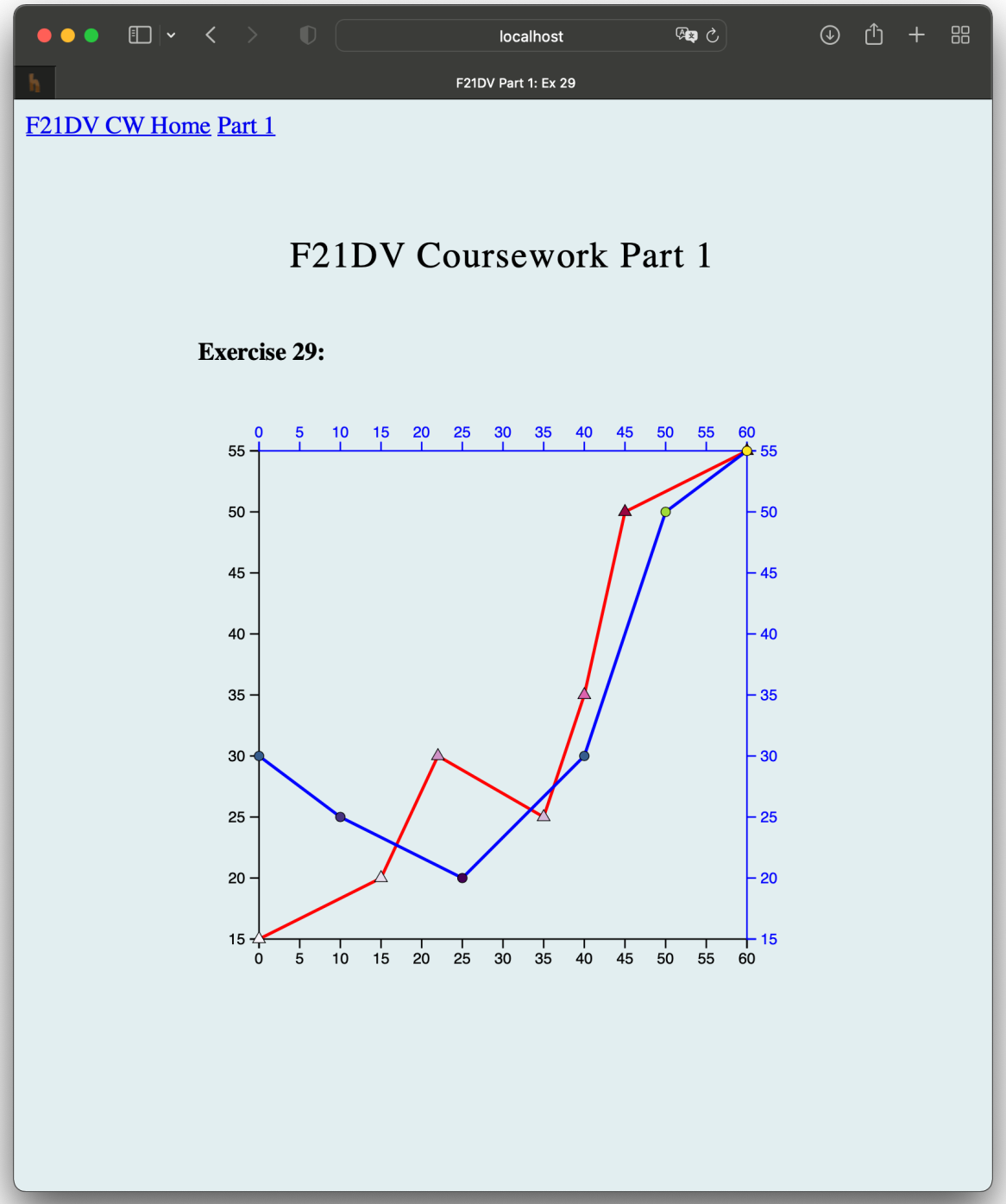

Figure 2.22: Exercise 29

This is essentially the same as exercise 25 and 26. The difference is instead of using `addLinesShape()`, we use `addLinesShapeDifferentColor()`. We now add a new parameter in the constructor of the function, defining the colour scheme to use.

2.24 Exercise 30 & 31

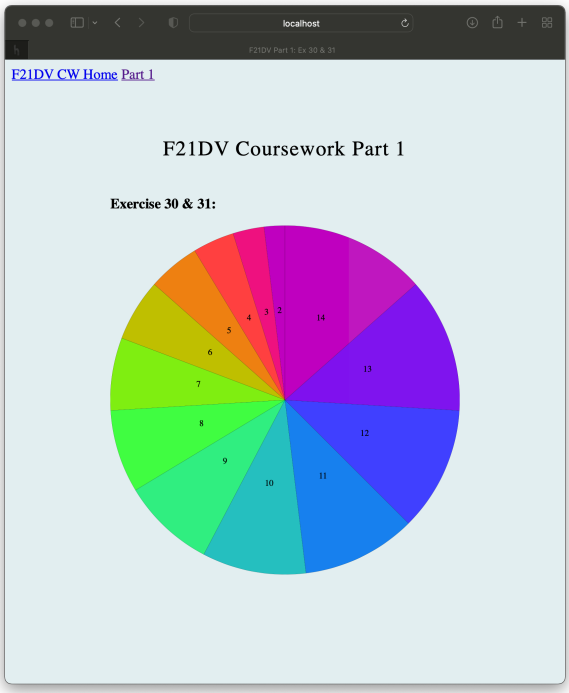

Figure 2.23: Exercise 30 & 31

```
1  // js script for part 1 Exercise:
2  const ex = '30 \& 31';
3
4  // Create Divs and button systematically using a general function.
5  import {createDiv} from '../functions.js';
6  createDiv(ex);
7
8  // Create and array of numebers. Remove '0'.
9  const [, ... data] = Array.from({length: 14}, (_, i) => i + 1);
10
11 // Svg dimension constants.
12 const xSize = 400, ySize = 400, margin = 40, xMax = xSize - margin*2, yMax = ySize -
       margin*2;
13
14 // Increasing the size of the answer container
15 d3.select('.container').style('height', `${ySize}px`);
16 d3.select('.answerCenter').style('width', `${xSize}px`);
17
18 // Svg Object.
19 const svg = d3.select('.answerCenter')
20                 .append('svg')
21                 .attr('width', xSize)
22                 .attr('height', ySize)
23                 .append('g')
24                 .attr('transform', `translate(${xSize/2}, ${ySize/2})`);
25
26 // Radius
27 const radius = Math.min(xSize, ySize) / 2;
28
29 // Color Scale
30 const colorScale = d3.scaleSequential()
31                         .domain(d3.extent(data))
32                         .interpolator(d3.interpolateSinebow);
33
```

```
34 // Create pi.
35 const pie = d3.pie();
36 const arc = d3.arc()
37                 .innerRadius(0)
38                 .outerRadius(radius);
39
40 // Generate Groups.
41 const arcs = svg.selectAll('arc')
42                 .data(pie(data))
43                 .enter()
44                     .append('g')
45                     .attr('class', 'arc')
46
47 // Draw arc paths.
48 arcs.append('path')
49     .attr('fill', (_, i) => colorScale(i))
50     .attr('stroke', 'black')
51     .attr('stroke-width', 0.1)
52     .attr('d', arc)
53
54 // Add arc's text.
55 arcs.append('text')
56     .text(d => d.data)
57     .attr('transform', d => `translate(${arc.centroid(d)})`)
58     .style('text-anchor', 'middle')
59     .style('font-size', 10)
```

../../public/js/part1/task30n31.js

The difference between this and earlier svgs is that this one scales using an additional arc scale. The shape that we append is now arc, we use the same scale to transform the text for each arc as well.

2.25 Exercise 32

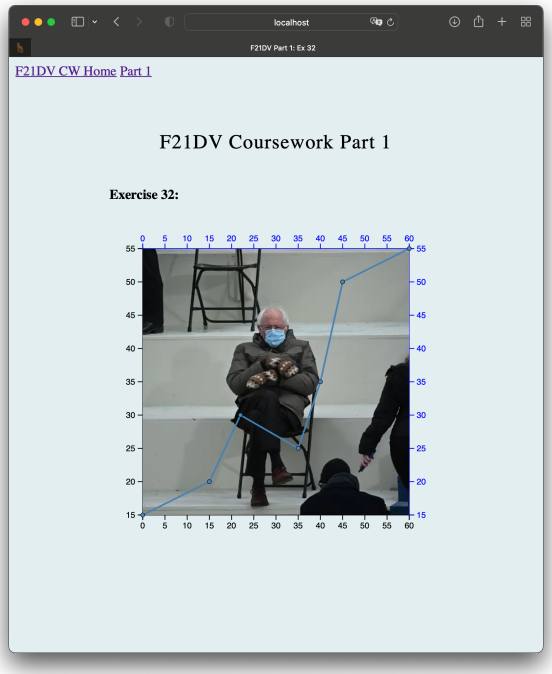

Figure 2.24: Exercise 32

```
// js script for part 1 Exercise:
const ex = 32;

// Import svg and line function from task22.js
import { createSvg, addLinesShape } from './task22.js';

// Import data for task 23.
const data = await d3.csv('../../data/part1/task23.csv');

// Append the svg element
const svg = createSvg(data, ex);

// Add the picture
svg.svg
    .append('svg:image')
    .attr('xlink:href', '../../data/part1/task32.png')
    .attr('width', 320)
    .attr('height', 320)

// Add line on top of pic.
addLinesShape(data, svg, 'steelblue', 'circle');
```

../../public/js/part1/task32.js

Now exercise 32 is pretty much an exercise 23. Difference is now we add an image before plotting the lines.