NEA Project Design

Jacob Halleron
The Sandon School

Summary

As stated in the analysis document, the project will be a dynamic website with a client-server model. On the server, there'll be a NodeJS app running the Express framework which will allow it to handle HTTP requests asynchronously. On instance of a request, the server will process it, and construct a webpage containing the relevant content. The client side is a traditional setup written in HTML, CSS and JavaScript. In essence, the browser will render a document with a diagram and the JS runtime will simultaneously continuously edit parts of the page if the correct buttons are selected.

In the "Proposed Solution" section of the analysis document, I've split the development of the project into six stages and loosely modelled this document around them. Some sections are much more complex and longer to plan than others, so they will have more documentation. Note that some parts of the project are partially designed but not in the project, I decided to trim down the project halfway through. Anything involving a tree or a graph was not properly implemented and exists in the `treejunk.js` or `graphjunk.js` files.

External technologies I'll use:

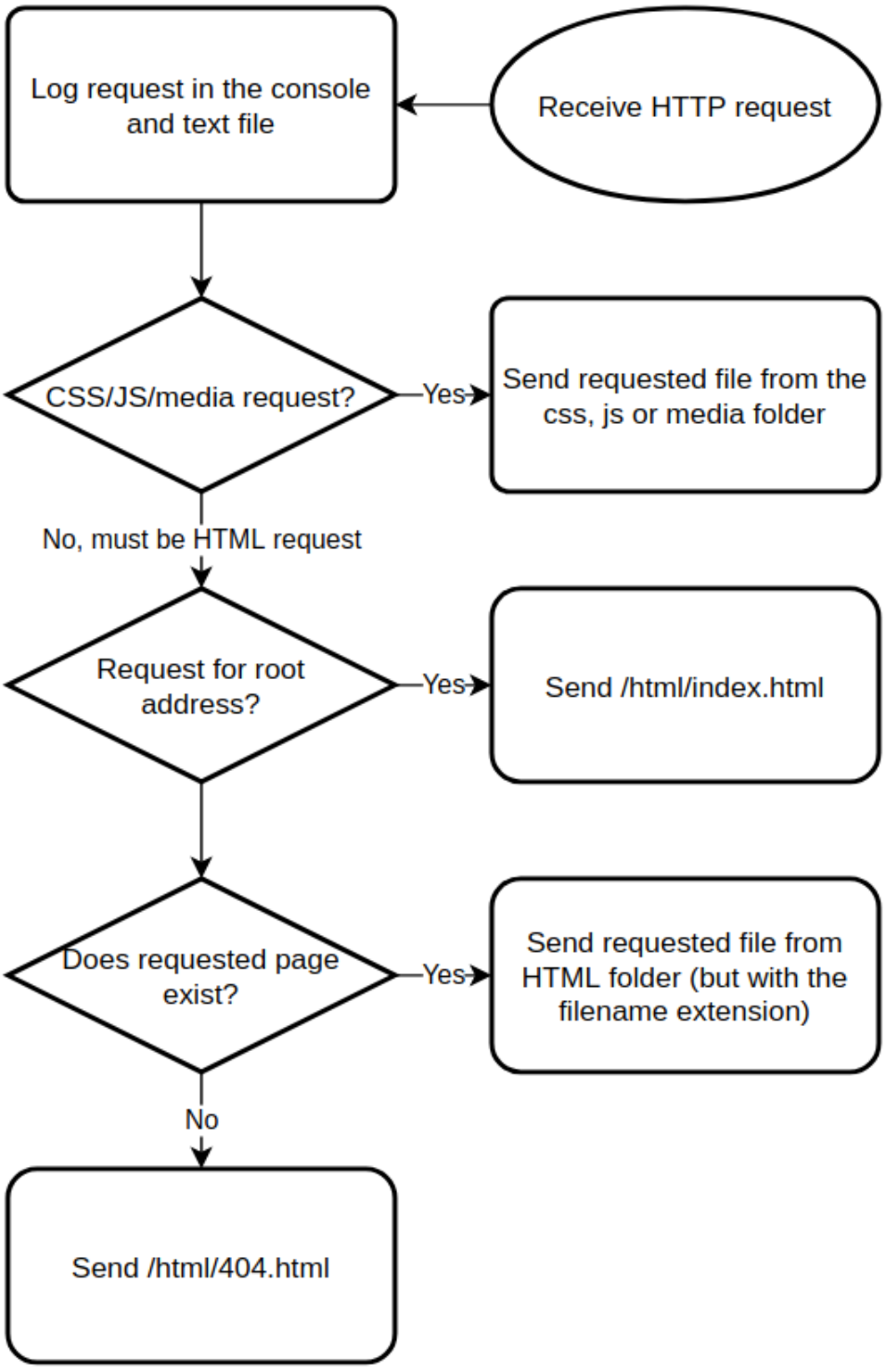
- HTML&CSS
- Javascript (including the DOM, canvas and audio APIs)
- NodeJS
- Express web server
- EJS templating module, for dynamic pages
- Web browser
- Git & Github

Server

The Express framework is a lightweight extension to the vanilla NodeJS HTTP module, and it doesn't add much other than make the server app source code slightly simpler. I was originally going to use the Apache web server with some basic templating coded in PHP, but my knowledge in PHP is limited at best. I know JS much better. It would be faster and more applicable to real life, though; most web servers have this setup along with MySQL or something.

The server, along with the client, is written in JavaScript, and one can define "routes" (responses triggered by requests to certain paths) in the core `app.js` file. The program is completely event-driven, so one need only define some functions, applicable to certain defined types of request whose inputs and outputs are instances of request and response objects (from the built-in classes).

The server will also log any requests it receives in the console and in a text file. Most web servers do this, it's considered good practice. I've put the whole server routing in a flow chart, shown in the figure.



Server flow chart

File Structure

In traditional NodeJS fashion, all files can be found in one directory hierarchy, shown here:

```
visualiser/
    doc/
        NEA Project Analysis.md
```

```
        NEA Project Documented Design.md
        NEA Project Technical Solution.md
        NEA Project Log.odt
        NEA Project Testing.md
    node_modules/
        ...
    src/
        css/
            main.css
        html/
            404.html
            index.html
            multiple.html
            single.html
        js/
            main.js
            algos.js
        app.js
        http.log
    package.json
    package-lock.json
    README.md
    start.sh
```

As you can see, everything is encapsulated in the `visualiser` folder, and all the code is in the `src` subfolder. `doc` contains all my documentation, including this file. `node_modules` is a directory automatically generated by NodeJS, which is necessary for importing external libraries. I won't directly interact with this. `css` is for stylesheet files (there should only be one) and `js` is for my scripts. I'll have two scripts: `main.js` for most classes and core site functionality, and `algos.js` for the algorithms themselves. The latter will get imported by the former. `app.js` is the back-end program for the server itself, it'll handle HTTP requests and respond with files from the CSS, HTML or JS folders. `http.log` is where HTTP requests will get logged by the server. `package.json` and `package-lock.json` are also autogenerated by NodeJS, they contain info about the project package. `README.md` is a file usually included in Git repositories, it's just a small description of the project. `start.sh` is a small startup script for quicker usage.

Web Design

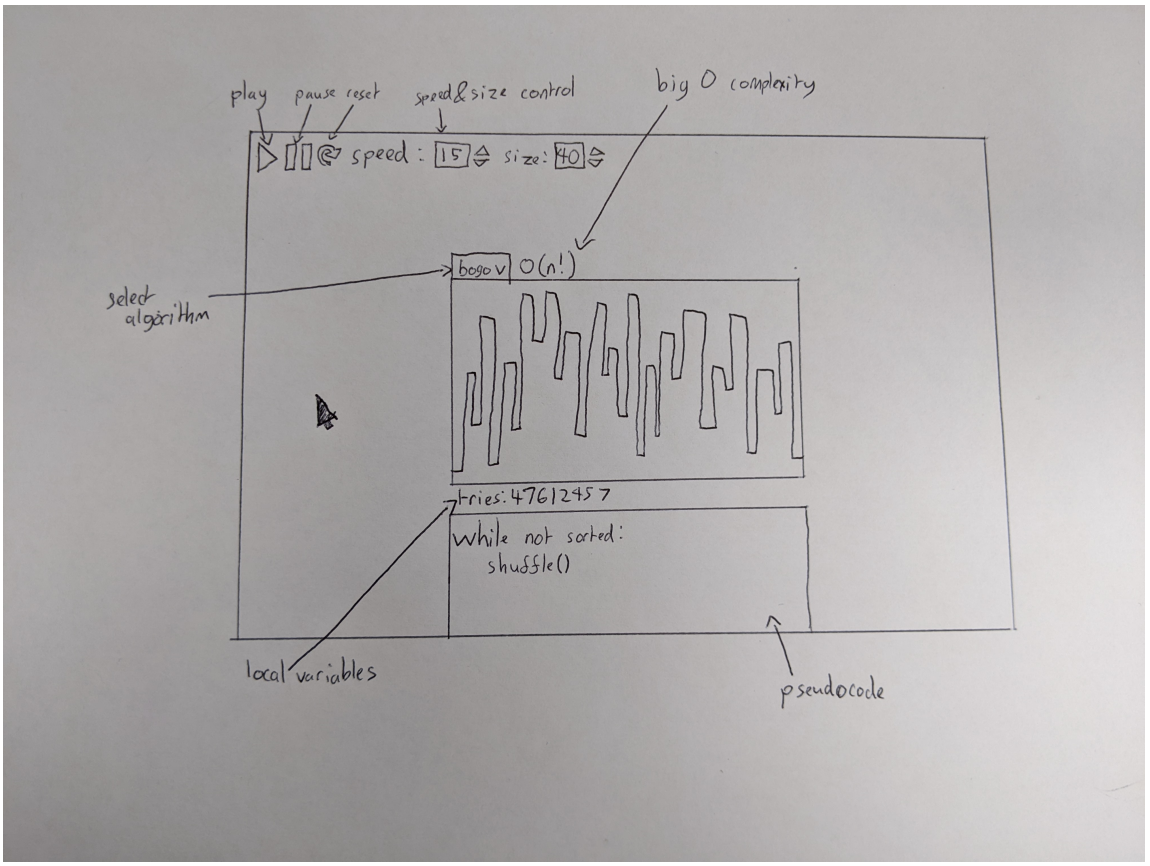
I'll make the site look simple and high contrast. The background is dark grey and text and parts of the chart is white. Designing custom buttons and icons is time-consuming and a lot of effort, but largely unnecessary so I'll just go with default styled buttons and text rather than icons.

There will be a few pages to design:

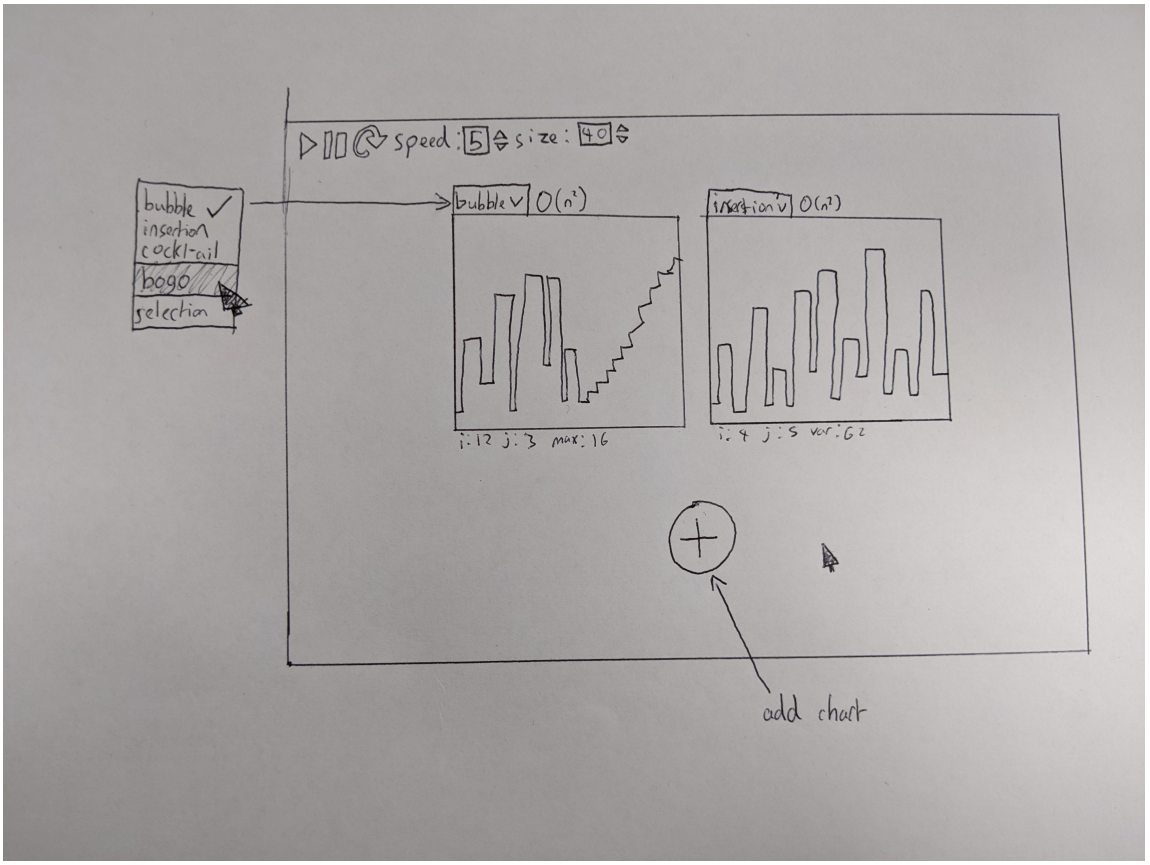
- About (homepage)
- Single animation page
- Multiple animation page
- 404 error page

Some of the HTML will inserted into the page by the program, so it can go as a string in the script rather than in the HTML files.

Here are some hand-drawn wireframe illustrations of the user interface:



This one's the "single animation" page, note that the pseudocode is visible and the chart is larger than on the multiple page.



Here's the page for multiple animations. The charts are smaller and there's the option to add another one.

The homepage is just going to be a heading, some text and a list of hyperlinks, which is not worth drawing a wireframe. The 404 page will just say 404: page not found.

Client Side JS

Most of the code in this project resides in the browser's JavaScript runtime.

Fundamental to this project is a box which shows the data structure being focused on. It'll be a completely resizeable canvas element and look different depending on the type of object. The diagram inside it will be either a bar chart (for sort and search algorithms) or a graph. Every time the running algorithm changes some variable, the box will be updated. Below are all the global variables I'm using in the script.

Variable	Type	Explanation
`actx`	Audio context	The "audio context", or sound interface, I'll use for any beeps and boops the program makes.

Variable	Type	Explanation
`GraphNode`	Class	Class for a single node of a graph. It has four properties: `id` which is a number unique to all the other nodes in a graph, `to` which is an array of the IDs of the other nodes it connects to, and `x` and `y` which are coordinates in the chart.
`TreeNode`	Class	Subclass of `GraphNode`, specifically for a tree structure. Also has property `from` which is the ID of its parent.
`Chart`	Class	Class for any chart
`SortChart`	Class	Class for a numeric array chart
`GraphChart`	Class	Class for graph structure chart
`TreeChart`	Class	Class for a tree structure chart
`algos`	Map	Stores all algorithms any chart might use. Each one is represented as an object with two methods, one which executes at the start (`init`), and one which executes every step after (`step`). The reasoning for this unorthodox algorithm representation will be explained later.
`charts`	Map	Contains all the charts currently on the screen

This “visualiser” will be represented as a new object, of the `Chart` class. Because there may be multiple charts running, I could have multiple of these objects at a given time. Depending on the object it represents, there will be different child class. The properties of this parent class are shown below.

Property	Type	Explanation
`id`	Number	Unique ID of this chart
`draw`	Function	The method to draw the object to the respective canvas. This will be executed every frame when running and will be different for different subclasses.
`play`	Function	Starts algorithm loop using my `setSpeed` method and stores the loop ID in `running`.

Property	Type	Explanation
		Also starts the loop which triggers the draw function each frame.
`pause`	Function	Stops algorithm loop.
`setAlgo`	Function	Changes the currently running function to the one with the given name in the global `algos`.
`setSpeed`	Function	Stops the running algorithm and restarts with the given speed.
`reset`	Function	Stops the running algorithm and randomises `value` using an appropriate algorithm depending on the subclass.
`running`	Number	The ID of the function currently running on this chart. The `setInterval` function regularly executes a given function at a certain rate, and returns a unique number so that this process can be cancelled.
`value`	Array or Tree	The most important property, the object being represented in the chart. For a bar chart this is a numeric array, for a graph or tree this is an array of nodes.
`shownValue`	Array or Tree	The value currently rendered in the actual chart; this may differ from `value` in the small time between the property access and drawing the frame. If `value` is too large this will not be used, for performance.
`algo`	String	Name of the current algorithm for this chart. This is also the key to find the relevant function in the global `algos` map.
`interval`	Number	Interval in milliseconds to wait between cycles of the algorithm.
`scanning`	Array	The index of whatever node or bar in `value` is being “scanned”, i.e. being looked at by the currently running algorithm. This will be drawn as a different colour to the other items.

Property	Type	Explanation
`done`	Boolean	Represents if the algorithm is done running, like if the bar chart has been sorted.

Here's a flow chart for the `play` method, it's one of the larger methods:

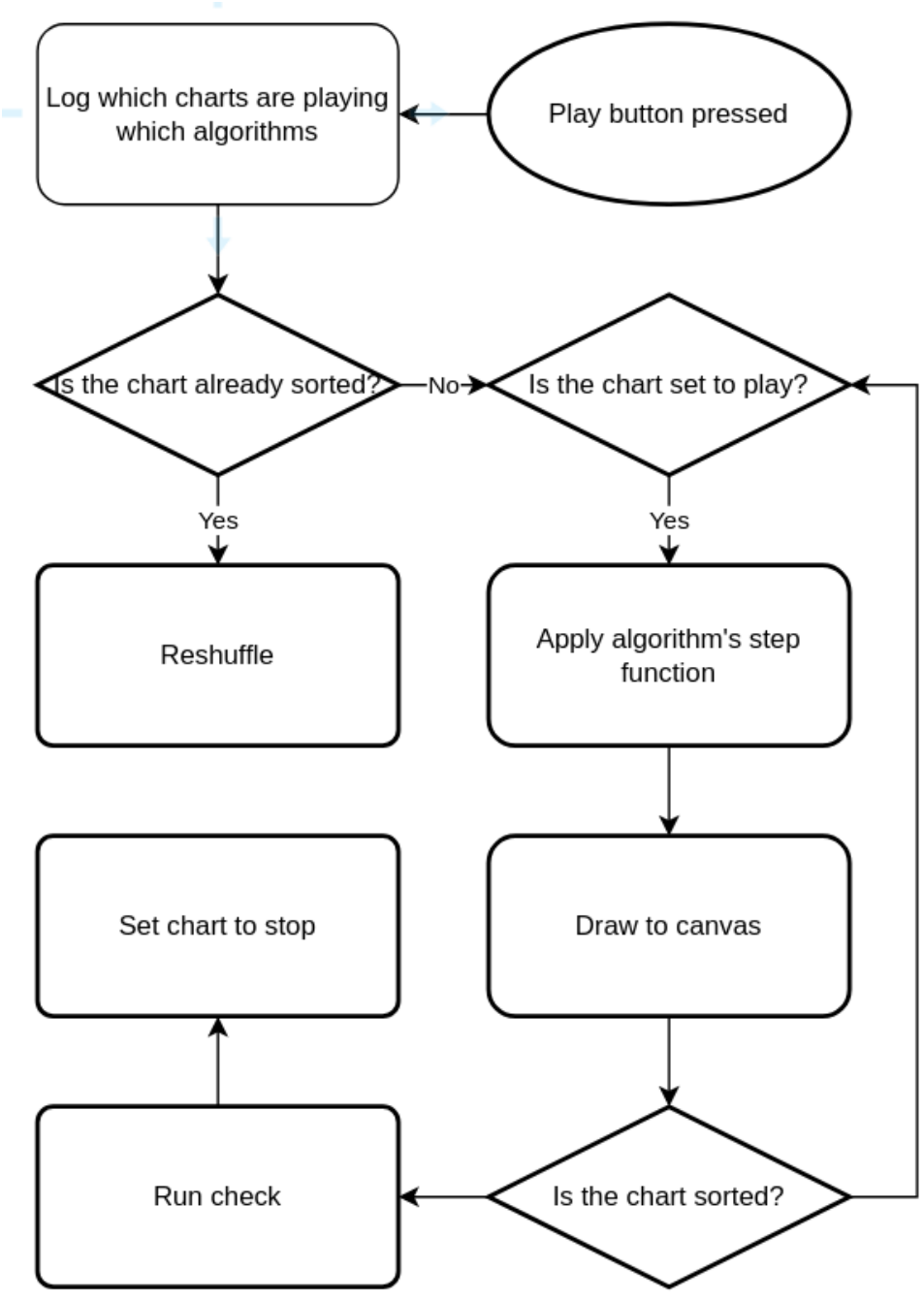


Chart `play` method flow chart

And here's some pseudocode for the `draw` method, flow charts aren't great to represent for loops:

```
if sound enabled:
    beep()
for each variable:
```

```
    write key and value to HTML list
clear canvas
bar width := chart width / array length
min bar height := chart height / array length
for each array element:
    colour := grey
    if element has moved since last draw:
        colour := white
    if chart is being checked:
        if element is in correct place:
            colour := green
    if element is being "scanned" by algorithm:
        colour := red
    bar height := min bar height * element's value
    draw bar
return
```

Bar Chart

If an array of numbers is inputted, the function will draw a bar chart, with each item's array index on the X-axis and its value on the Y-axis. There won't be labels or axis lines though, they won't be necessary to understand the process of the algorithm. Each bar will be coloured according to different factors, e.g. the currently scanned bar being red, recently moved ones being lighter than others, bars being green when sorted, etc. Underneath the bar chart may be one or more arrows to indicate certain variables related to the array in the running algorithm. The client machine will produce beep and boop sounds with pitch varying with the height of the bar being scanned.

This class of visualiser will have some of its own variables:

Property	Type	Explanation
`swap`	Function	Swaps numbers at the two array index arguments
`setLength`	Function	Sets the length of the array.
`beep`	Function	Makes a beep sound with pitch corresponding to the inputted bar height.
`oscillator`	Oscillator	"Oscillator" object from the built-in sound library for beeps, boops and such

Property	Type	Explanation
`actx`	Audio context	Audio context to control the sounds the chart makes. This sound library requires a lot of variables.

The draw function will also be specific to this class. Rendering will be done by drawing rectangles on the canvas of uniform width, and height equal to that number's index divided by the array length, multiplied by the canvas height. If the chart's `interval` property is more than a certain amount of milliseconds, this function will utilise the `shownValue` property to create a short animation for any bars that have moved.

Graph

Finally, if the input is a graph structure, a graph (in the technical mathematical sense) will be drawn on the canvas. A graph is effectively an array of nodes, with each node connected to others. I'll make another class for a graph node:

Property	Type	Explanation
`id`	String	The unique identifier of this node, also probably just one letter long.
`x`	Number	X coordinate of this node on the canvas.
`y`	Number	Y coordinate of this node on the canvas.
`to`	Array	Array containing the IDs of any other nodes this leads to.

Graph drawing is a surprisingly wide branch of computer science, and the most popular method of formulating a diagram is through a method known as force directed graph drawing. Essentially, the system is simulated as if each vertex has forces applied to it that attract or repel connected vertices. After not too long, there is equilibrium. This technique is used by advanced and professional statistics software, so it might be overkill for the small examples in this project, but it looks fun to program so I'll implement it.

Tree

If a tree object is inputted (a binary or nonbinary tree), a diagram with nodes and edges will be drawn in the box. A node will be a simple circle with a letter

or number inside, and an edge will be a line. I'll make a subclass of graph for it:

Property	Type	Explanation
`id`	String	The unique identifier of this node, probably just one letter long.
`to`	Array	Array containing the IDs of any children nodes.
`x`	Number	X coordinate of this node on the canvas.
`y`	Number	Y coordinate of this node on the canvas.
`mod`	Number	Used in the drawing algorithm to calculate how how much to modify its X coordinate.
`traverse`	Function	Accepts one argument, a string which is either "post" or "pre". Returns an array of the values of the appropriate traversal of this node and its children. In order traversal will only be possible for binary trees, which this may not be.

The Y position of a given node will depend on its depth in the tree, i.e. the root node goes at the top and the deepest leaf at the bottom, with nodes with equal depth having equal Y positions The X position, however, will be more tricky to calculate; I'll want parent nodes centered above their children (so there aren't more nodes under one side than the other) and all the nodes relatively evenly distributed so that we don't waste space on the screen or cram too many into a small space. This isn't *technically* necessary, but will

make the diagram more aesthetically pleasing and easier for the user to scan, thereby improving the user experience. This article and its attached source code have been especially useful for the tree-drawing process, and I'll be using an algorithm similar to the one provided in it. It involves initially spacing the children apart, then centering the parents over them, then making sure there are no collisions in their positions. Here's its pseudocode, it's surprisingly lengthy:

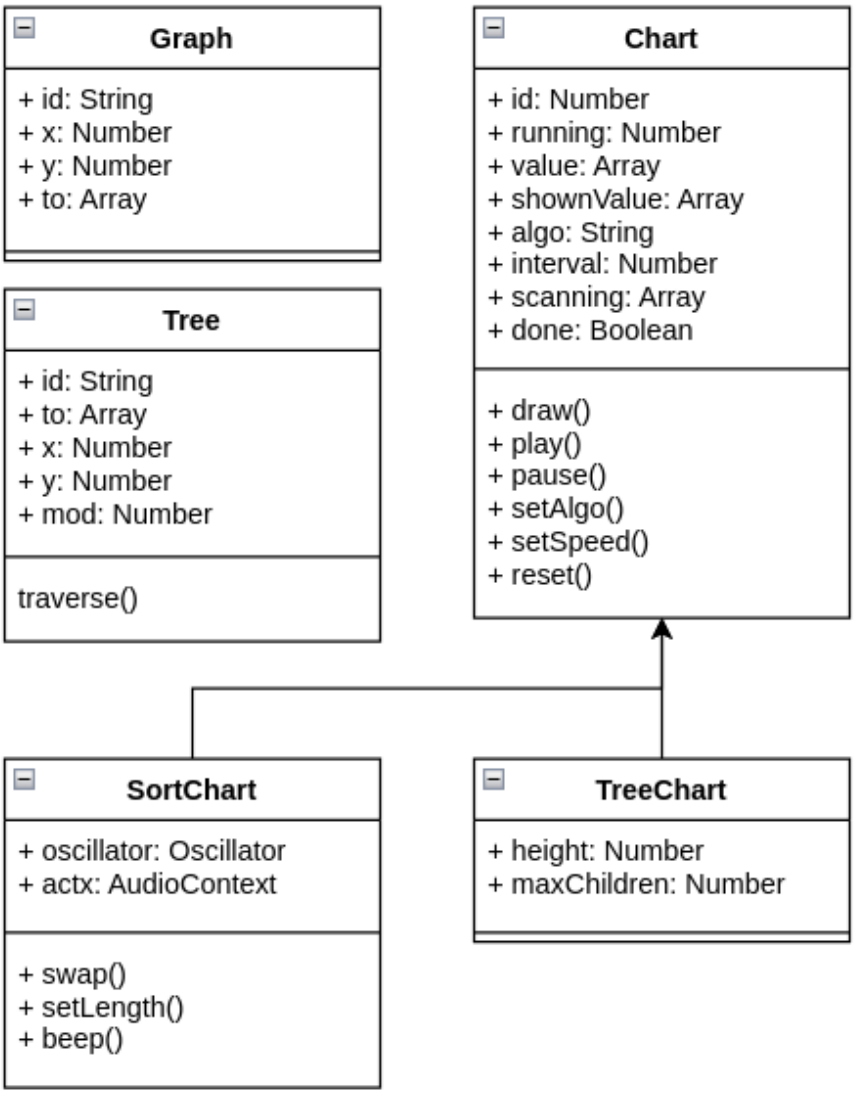
```
function calculateInitialX(tree):
    for child in tree.children:
        calculateInitialX(child)
    if tree has left siblings:
        tree.x = tree.leftSibling.x + 1
    if tree has no children:
        tree.x = 0
    if tree has 1 child:
        if tree has left siblings:
            tree.mod = tree.x – tree.children[0].x
        else:
            tree.x = tree.children[0].x
    if tree has multiple children:
        mid = (tree.children[0].x + tree.children[last].x) / 2
        if tree has left siblings:
            tree.mod = tree.x - mid
        else:
            tree.x = mid
    if tree has children and left siblings:
        conflictCheck(tree)

function conflictCheck(tree):
    # mindistance = 1
    shiftValue = 0
    contour = getLeftContour(tree) # minimum x value of children
    sibling = tree.leftmostSibling
    while sibling exists and is not tree:
        siblingContour = getRightContour(sibling)
        for level = tree.y to minimum(contour.keys):
            if (contour[level] - siblingContour[level]) < shiftValue
                shiftValue = 1 – distance
        if shiftValue > 0:
            tree.x += shiftValue
            tree.mod += shiftValue
            centerNodesBetween(tree, sibling)
            shiftValue = 0
        sibling = sibling.rightSibling

function calculateFinalPositions(tree):
    tree.x += tree.mod
```

```
    for child in tree.children:
        calculateFinalPositions(child)
```

As with the bar and graph charts, I'll make another subclass for the tree chart:

Property	Type	Explanation
`height`	Number	How many nodes tall (deep) the tree will be after generation.
`maxChildren`	Number	The maximum number of children a node can have for generation.

Here's a UML diagram for all the classes I've designed:



UML Class Diagram

Algorithms

The algorithms themselves will not be as simple as rewriting pseudocode from the internet. Integral to the design of the site is the ability to start and stop the algorithm at will, and change the speed it runs. JavaScript only has functionality to execute statements at the maximum speed possible, or trigger a function regularly at a set interval. The simplest solution I can think of is to convert all the code for an algorithm into a function that will repetitively be executed for every step of the process.

Each algorithm will be represented as one function that is only applied at the start, and another which is executed for every step of the process. These functions will be called by the chart object using using the `apply()` method of the Function class. `apply()` calls the function using a specific given scope, i.e. you can call an external function as if it were a method. Using this, an algorithm can access the "host" object's private methods and properties.

I've figured out a process for making these algorithms doable, as long as they don't use recursion. All the algorithms contain either while loops or for loops, and these aren't too complex to convert to my single function format. Here are some pseudocode examples:

While loop:

```
if foo:
    bar()
else:
    break()
```

For loop:

```
i = 0 // in the init function
if i < foo:
    bar()
    i++
else:
    break
```

If you have a nested loop, you can convert that too by putting the extra steps around the inner loop, in the inner `else` block, e.g:

```
while i:
    a()
    while j:
        b()
```

```
    c()
```

...becomes...

```
a() // in the init function
if i:
    if j:
        b()
    else:
        c()
        a()
```

The algorithms I'll implement are as follows. `a` means the inputted array and `t` means the index of the target for search algorithms. In graph algorithms, `g` is the graph and `v` is the starting vertex.

Bar chart:

- Bubble sort

```
while not sorted:
    for i = 0 to len(a) - 1:
        if a[i] > a[i + 1]:
            swap(i, i + 1)
```

- Bogo sort

```
while not sorted:
    shuffle(array)
```

- Insertion sort

```
i = 1
while i < len(a):
    j = i
    while j > 0 and a[j - 1] > a[j]:
        swap(j, j - 1)
    i++
```

- Selection sort

```
for i = 0 to len(a) - 1:
    jMin = i
    for j = i + 1 to len(a):
        if a[j] < a[jMin]:
```

```
            jMin = j
    if jMin != i:
        swap(i, jMin)
```

Tree:

- Pre order

```
visit(root)
for each child ascending:
    traverse(child)
```

- In order

```
traverse(left)
visit(root)
traverse(right)
```

This one only works with binary trees, the other two work with trees of any number of children.

- Post order

```
for each child ascending:
    traverse(child)
visit(root)
```

- Binary search

```
l = 0
r = len(a) - 1
while l <= r:
    m = floor((l + r) / 2)
    if a[m] < t:
        l = m + 1
    else if a[m] > t:
        r = m - 1
    else:
        return m
```

Graph:

- Depth first search

`s` is a stack.

```
s.push(v)
while s not empty:
    v = s.pop()
    if not v.visited:
        v.visited = true
        for all edges from v to w in g.adjacentEdges(v):
            s.push(w)
```

- Breadth first search

`q` is a queue.

```
v.visited = true
q.enqueue(v)
while q not empty:
    v = q.dequeue()
    if v is goal:
        return v
    for all edges from v to w in g.adjacentEdges(v):
        if not v.visited:
            v.visited = true
            q.enqueue(w)
```

- Dijkstra

```
for i of g:
    i.dist = infinity
    i.prev = null
    q.enqueue(v)
v.dist = 0
while q not empty:
    u = vertex in q with min dist
    q.dequeue
    for i of u still in q:
        alt = u.dist + len(u, i)
        if alt > v.dist:
            v.dist = alt
            v.prev = u
```