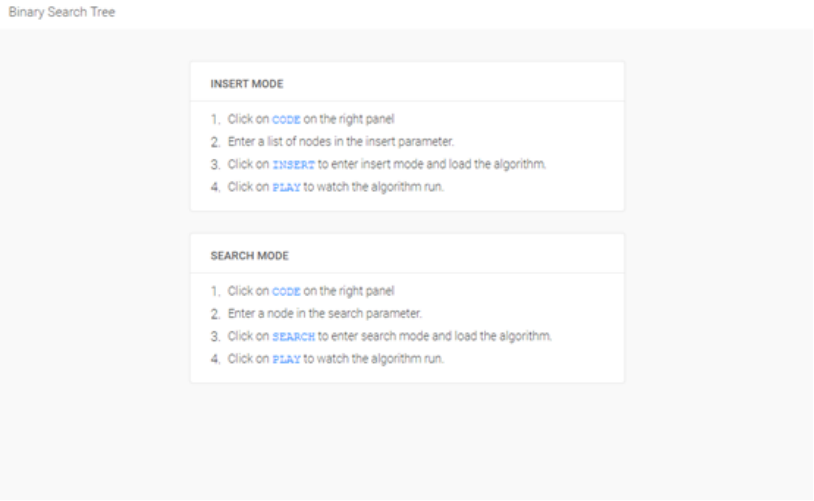
User Guide: Change Content

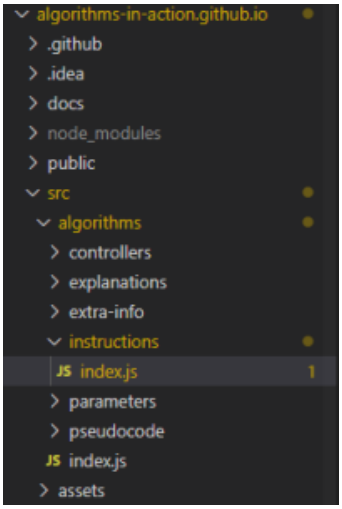
This guide walks through how to change text for different sections in the UI.

Modify “Instructions”

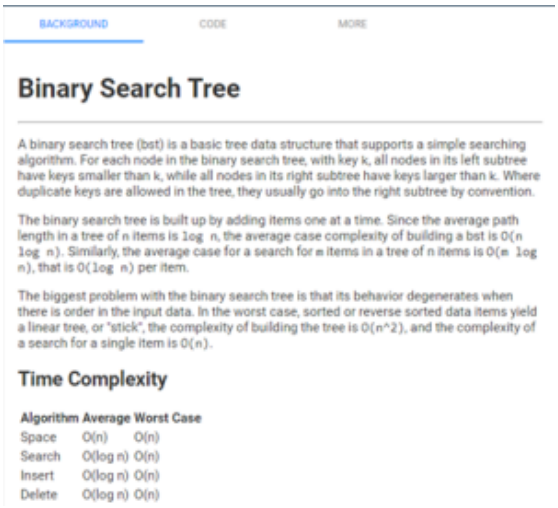
1. Modify "Instruction"



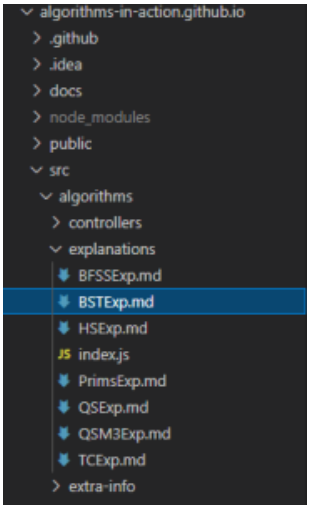
Go to the following file and change the instruction’s text.



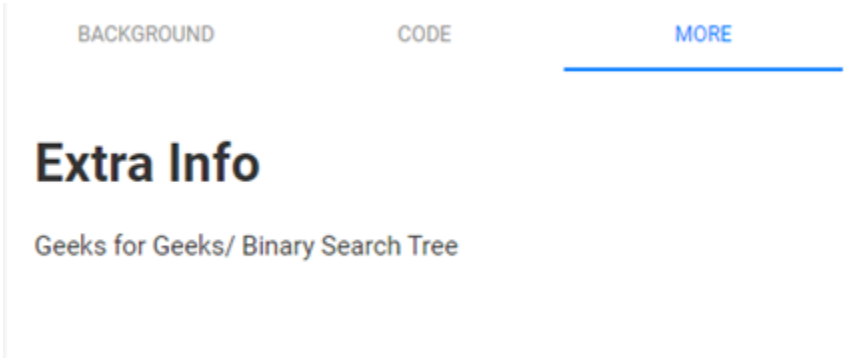
2. Modify "Background" Tab



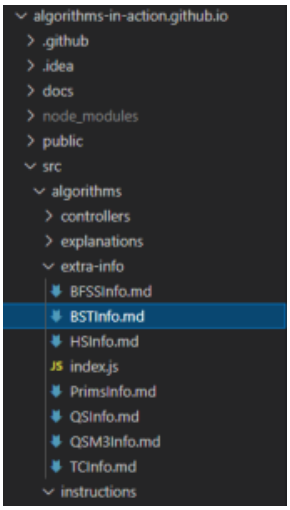
Go to the following directory and modify the associated markdown file.



3. Modify "More" Tab



Go to the following directory and modify the associated markdown file.


algorithms-in-action.github.io
.github
.idea
docs
node_modules
public
src
algorithms
controllers
explanations
extra-info
BFSSInfo.md
BSTInfo.md
HSInfo.md
index.js
PrimsInfo.md
QSInfo.md
QSM3Info.md
TCInfo.md
instructions

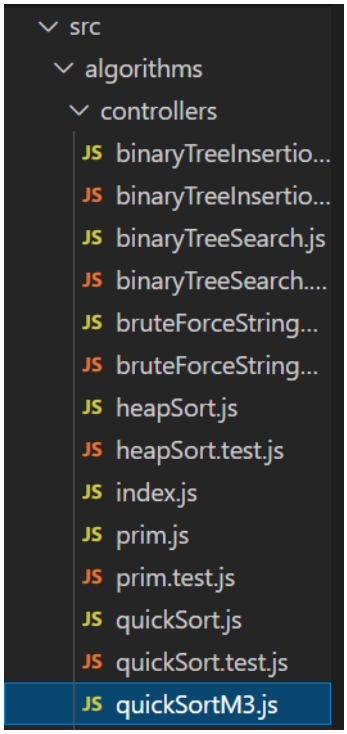
User Guide

This guide walks through how "Quicksort - Median of Three" and "Brute Force String Search" were added to the app by the current team. A general guide about how to add algorithms can be found here https://github.com/algorithms-in-action/algorithms-in-action.github.io/wiki/Development-Manual

1. Quicksort - Median of Three:

Steps followed for adding Median of Three as a new algorithm:

1. 'quickSortM3.js' file added under the folder 'src/algorithms/controllers'.

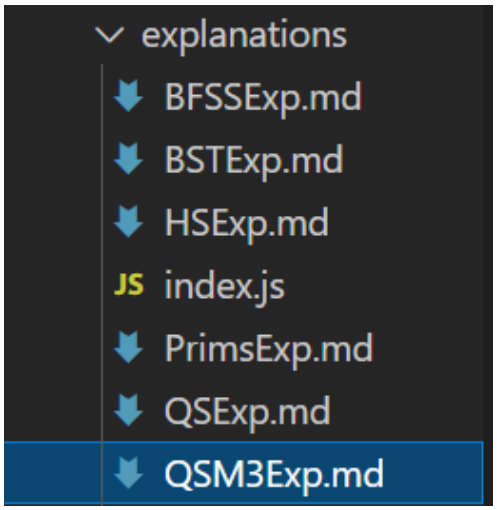


2. Registered the new file in 'src/algorithms/controllers/index.js'.

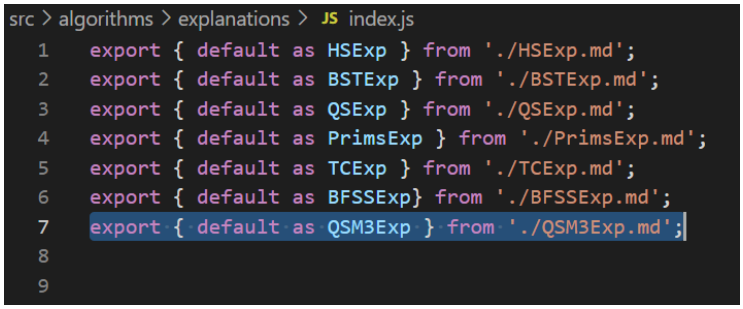
src > algorithms > controllers > index.js

```
1   export { default as binaryTreeSearch } from './binaryTreeSearch';
2   export { default as binaryTreeInsertion } from './binaryTreeInsertion';
3   export { default as heapSort } from './heapSort';
4   export { default as quickSort } from './quickSort';
5   export { default as prim } from './prim';
6   export { default as transitiveClosure } from './transitiveClosure';
7   export {default as bruteForceStringSearch} from './bruteForceStringSearch';
8   export {default as quickSortM3} from './quickSortM3';
9
```

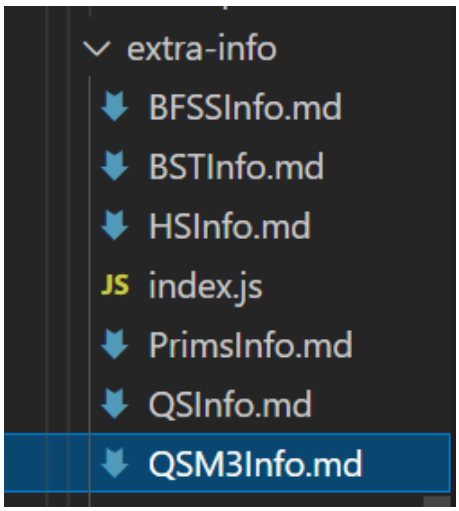
3. 'QSM3Exp.md' file added under the folder 'src/algorithms/explanations'.



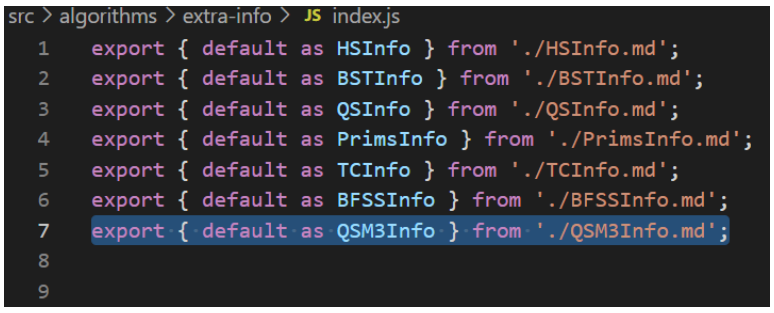
4. Exported the new file in 'src/algorithms/explanations/index.js'.



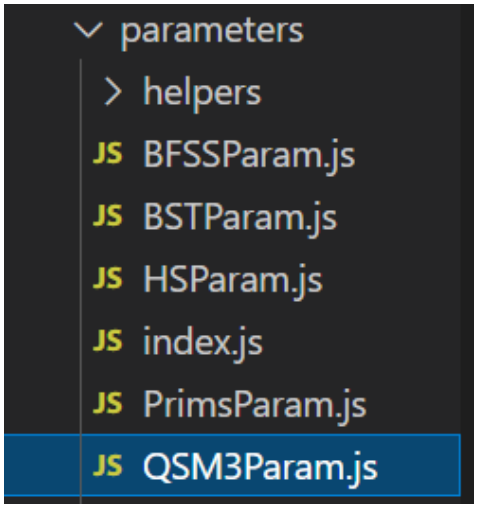
5. 'QSM3Info.md' file added under the folder 'src/algorithms/extra-info'.



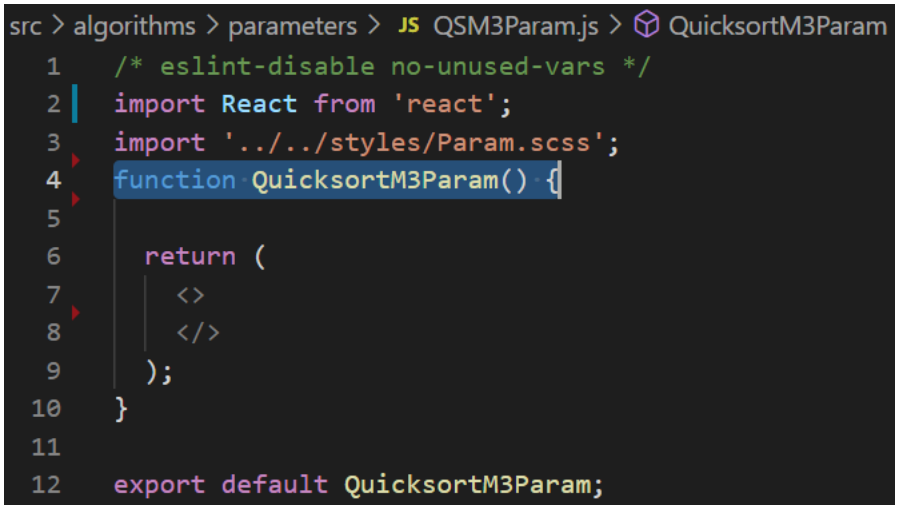
6. Exported the new file in 'src/algorithms/extra-info/index.js'.



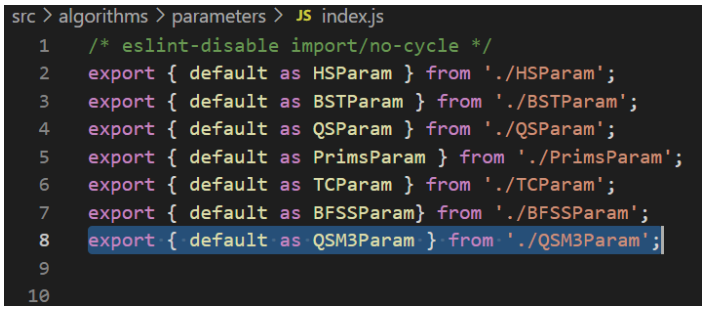
7. 'QSM3Param.js' file added under the folder ‘src/algorithms/parameters’.



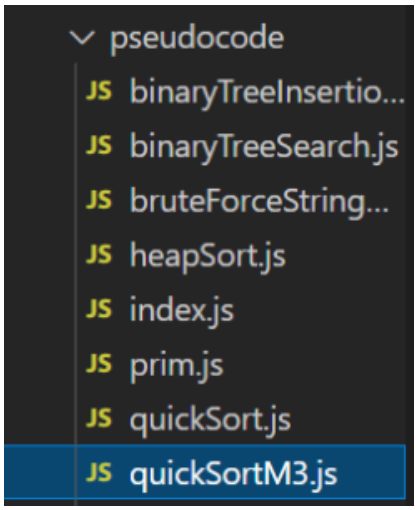
8. Initialized the 'QSM3Param.js' file in the following format.



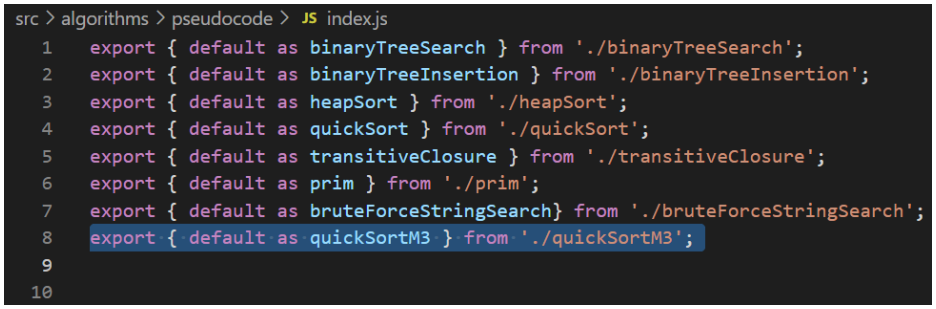
9. Exported the new file in 'src/algorithms/parameters/index.js'.



10. 'quickSortM3.js' file added under the folder ‘src/algorithms/pseudocode’.

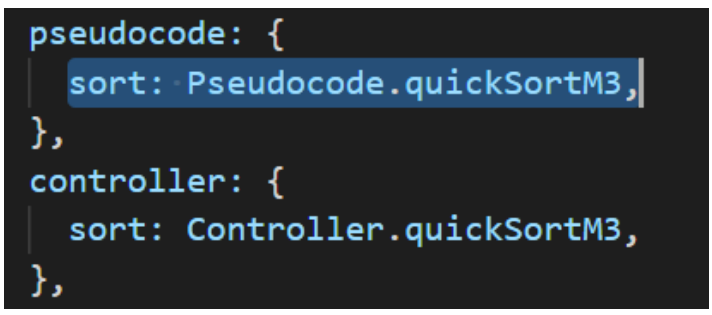


11. Exported the new file in 'src/algorithms/pseudocode/index.js'.



12. New algorithm added under the folder 'src/algorithms/index.js' in const ‘algorithms’ by the following format.

13. Controller and pseudocode used by the following format.



14. Category value set as the category's name.

```
name: 'Quicksort (Median of 3)',
category: 'Sorting',
```

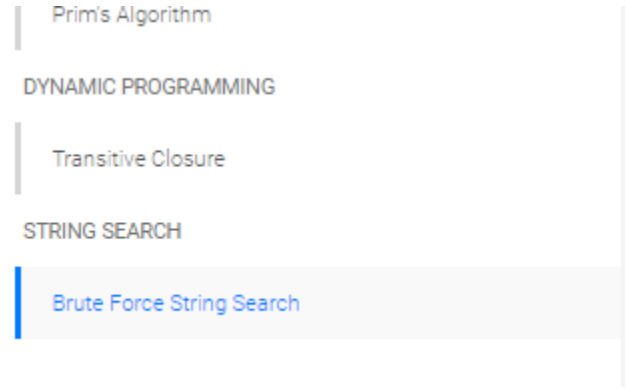
2. Brute Force String Search:

Steps followed for adding Brute Force String Search as a new algorithm:

Writing BruteForceStringSearch.js:

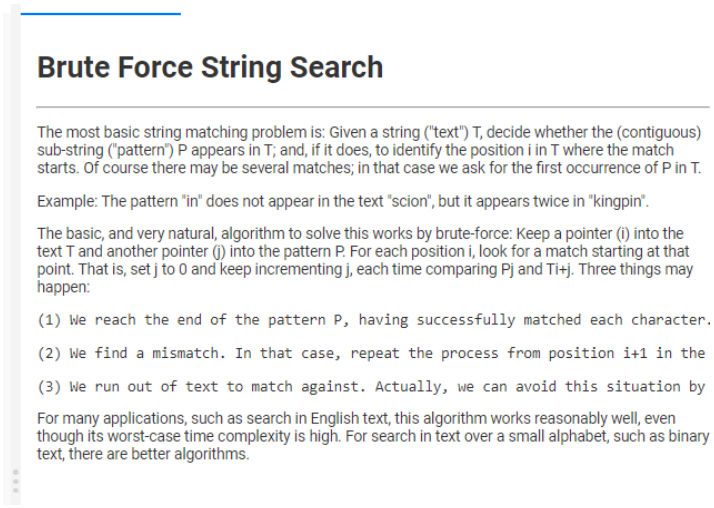
algorithms/index.js controls what it passed to the algorithm All of the items listed below must be created. "name" and "category" are what are displayed in the left hand navbar.

```
'bruteForceStringSearch': {
  name: 'Brute Force String Search',
  category: 'String Search',
  explanation: Explanation.BFSSExp,
  param: <Param.BFSSParam />,
  instructions: Instructions.BFSSInstruction,
  extraInfo: ExtraInfo.BFSSInfo,
  pseudocode: {
    search: Pseudocode.bruteForceStringSearch,
  },
  controller: {
    search: Controller.bruteForceStringSearch,
  },
},
};
```



explanations/BFSSExp.md is a text file that displays under the background panel.

```
# Brute Force String Search
---

The most basic string matching problem is: Given a string ("text") T, decide whether the (contiguous) sub-string
("pattern") P appears in T; and, if it does, to identify the position i in T where the match starts. Of course there
may be several matches; in that case we ask for the first occurrence of P in T.

Example: The pattern "in" does not appear in the text "scion", but it appears twice in "kingpin".

The basic, and very natural, algorithm to solve this works by brute-force: Keep a pointer (i) into the text T and
another pointer (j) into the pattern P. For each position i, look for a match starting at that point. That is, set j
to 0 and keep incrementing j, each time comparing P[j] and T[i+j]. Three things may happen:

    (1) We reach the end of the pattern P, having successfully matched each character. In that case we report success:
    return the pointer to the start of the match, that is, i.

    (2) We find a mismatch. In that case, repeat the process from position i+1 in the text.

    (3) We run out of text to match against. Actually, we can avoid this situation by not letting i run all the way to
```



Brute Force String Search

The most basic string matching problem is: Given a string ("text") T, decide whether the (contiguous) sub-string ("pattern") P appears in T; and, if it does, to identify the position i in T where the match starts. Of course there may be several matches; in that case we ask for the first occurrence of P in T.

Example: The pattern "in" does not appear in the text "scion", but it appears twice in "kingpin".

The basic, and very natural, algorithm to solve this works by brute-force: Keep a pointer (i) into the text T and another pointer (j) into the pattern P. For each position i, look for a match starting at that point. That is, set j to 0 and keep incrementing j, each time comparing Pj and Ti+j. Three things may happen:

```
(1) We reach the end of the pattern P, having successfully matched each character.

(2) We find a mismatch. In that case, repeat the process from position i+1 in the

(3) We run out of text to match against. Actually, we can avoid this situation by
```

For many applications, such as search in English text, this algorithm works reasonably well, even though its worst-case time complexity is high. For search in text over a small alphabet, such as binary text, there are better algorithms.

parameters/BFSSParam.js creates the form that accepts input. The structure of the project only allows for one input per algorithm. To work on this, the files that must be edited are probably StringParam and ParamForm.js

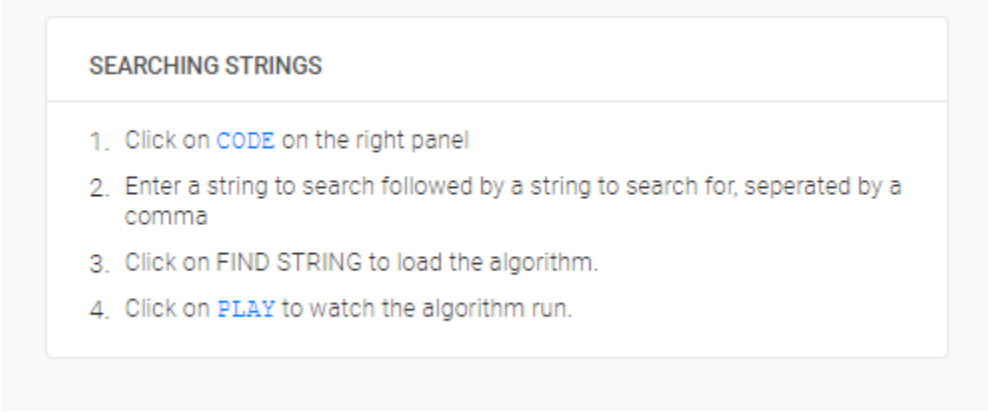
```
return (
  <>
    <div className="form">
      <StringParam
        name="bruteForceStringSearch"
        buttonName="Search"
        mode="search"
        formClassName="formLeft"
        DEFAULT_VAL={array}
        SET_VAL={setArray}
        ALGORITHM_NAME={BFST_SEARCH}
        EXAMPLE={BFST_EXAMPLE}
        setMessage={setMessage}
      />
    </div>
    {/* render success/error message */}
    {message}
  </>
);
```

helpers/StringParam.js was created to allow string input. It is a copy of helpers/ListParam that uses a different regex and method to parse input to allow strings. Regex wasn't working so we just used true. Regexes are kept under ParamHelper.js

```
const handleDefaultSubmit = (e) => {
  e.preventDefault();
  const inputValue = e.target[0].value.replace(/\s+/g, '');
  if (true/*stringListValidCheck(inputValue)*/) {
    const nodes = inputValue.split`,`.map((x) => ""+x);
    // SET_VAL(nodes);
    // run animation
    dispatch(GlobalActions.RUN_ALGORITHM, { name, mode, nodes })
    setMessage(successParamMsg(ALGORITHM_NAME));
  } else {
    setMessage(errorParamMsg(ALGORITHM_NAME, EXAMPLE));
  }
};
```

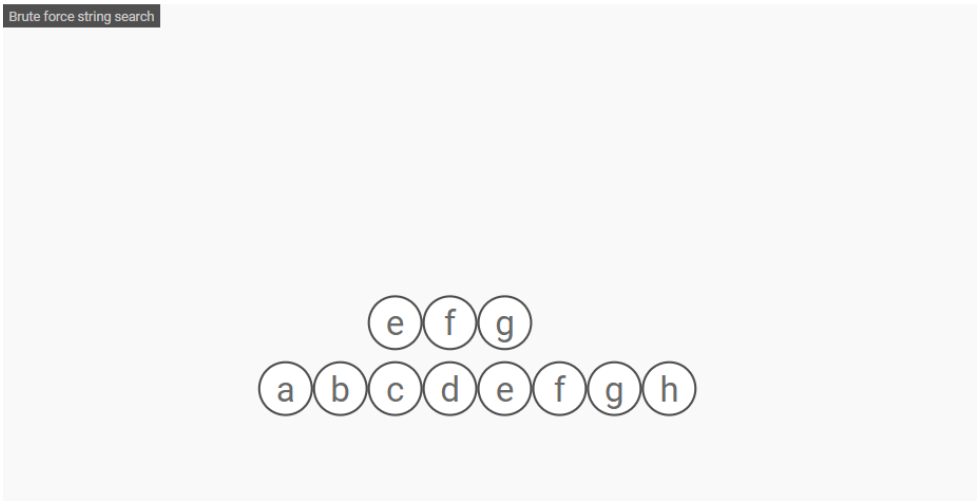
Instructions.BFSSInstruction is a constant kept under the Instructions/index.js file. This is the initial instruction presented to the user when the algorithm is acessed

```
const stringInstructions = [{
  title: 'Searching Strings',
  content: [
    `Click on ${KEY_CODE} on the right panel`,
    'Enter a string to search followed by a string to search for, seperated by a comma',
    `Click on ${KEY_FIND} to load the algorithm.`,
    `Click on ${KEY_PLAY} to watch the algorithm run.`,
],
}];
```



The BFSS Controller runs the bulk of the animation in conjunction with the chunker function. This file uses a graph instance as it's visualiser. Arrays were not chosen as it was difficult to get and manipulate multiple arrays within 1 JavaScript panel. Editing of the ArrayTracer file may be able to facilitate this in future implementations but could not be achieved in the Sprint 2 timeframe. It would also be nice if the shape of the array was altered to have it look like an array, and the colors for succesful matches changed.

```
initVisualisers() {
  return {
    graph: {
      instance: new GraphTracer('bst', null, 'Brute force string search'),
      order: 0,
    },
  };
},
```



Each step in the algorithm is linked using the chunker to the corresponding pseudocode. Each of these are 1 step in the progress bar.

```
if(searchString[shift_i+shift_j] != findString[shift_j]){
  chunker.add('3', (vis, i,j, n) => {
```

The pseudocode is kept in the pseudocode directory under bruteForceStringSearch.js. Bookmarks that the chunker access are denoted by \\B (number). This is the code that is run in the Code panel.

```
// Look for a match starting from T[i]
j <- 0
while j < m and P[j] = T[i+j] \\B 3
\\Expl{  We keep progressing the search as long as we have not
        exhausted the pattern (that is, j<m) and the pattern
        checked so far matches the string starting from T[i].
\\Expl}
```

There is currently a conflict in the GraphTracer.js file which must be resolved to allow the algorithm to play back.

```
BruteForceStringSearch(T, n, P, m)
// Look for pattern P (of length m) in text T (of length n).
// If found, return the index of P's first occurrence in T.
// Otherwise return -1.
    i ← 0
    while i+m <= n
//        Look for a match starting from T[i]
        // Look for a match starting from T[i]
        j ← 0
        while j < m and P[j] = T[i+j]
            j ← j+1
        if j = m
            return i
        i ← i+1
    return (-1)
```

All of these files should be exported with a suitable name in their corresponding index.js files. Example given for explanations file, but similar for all other files (basic react knowledge).