# 1 Homework 1 Solution

**Due Date**: Sep 24; To be turned in on paper in class.

**Important Reminder**: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

Please remember to justify all answers.

Note that some of the questions require you to show code. You may use a JavaScript implementation to verify your answers but you should realize that you will not have access to an implementation during exams.

You are encouraged to use the web or the library but are required to cite any external sources used in your answers.

**Restrictions**

Your answers to Questions 1 - 12 may not make any explicit use of destructive assignment, iteration or recursion. You may use any String or Array functions.

Questions 1 - 12 are meant to familiarize you with the built-in functions available in JavaScript (and many other languages) for arrays and strings. Some hints:

- In the absence of assignment and iteration, your function bodies will consist of a single return statement returning a single expression.
- Your functions cannot contain any statements other than the single `return` statement. In particular it cannot contain `if-else` statements, but can use conditional expressions involving the ternary operator `?:`.
- Use higher-order Array functions to replace the use of iteration.
- Note that the functions provided to many of the `Array` functions like `map()` and `reduce()` take multiple arguments.
- Look at the ways the `Array()` constructor can be called.
- The `Array.fill()` function may be useful for setting up initial arrays.
- To give you some idea of what is expected, here is a function which returns an array containing the first `n` factorials:

```
/** If n > 0, return an array arr of length n such
 *         that arr[i] === factorial(i) for all i < n
 */
function factValues(n) {
  return new Array(n-1).
    fill(0).
    map((_, i) => i + 2).
   reduce((acc, e, i) => acc.concat([e*acc.slice(-1)[0]]),
          [1]);
    }
```

We create an initial array of length `n - 1` and map its indexes to generate $2 \ldots n$; we reduce these mapped indexes with an accumulator (initialized to `[1]`) accumulating the values with the next value computed as the mapped index multiplied by the last value accumulated so far.

Note that instead of using `acc.slice(-1)[0]` to pick up the last acc value, we can use `acc[i]` instead.

1. Subject to the above restrictions, show code for a function `rmPrefixSuffix(str, m, n)` which, when given a string `str` and non-negative integers `m` and `n`, returns string `str` with the first `m` characters and last `n` characters removed. *3-points*

   ```
   > rmPrefixSuffix('Twas brillig and the slithy toves',
                    5, 10)
   'brillig and the sl'
   > rmPrefixSuffix('', 5, 10)
   ''
   rmPrefixSuffix('Twas brillig and the slithy toves',
                  0, 0)
   'Twas brillig and the slithy toves'
   ```

   This function can be implemented directly by simply using `substring()`:

   ```
   function rmPrefixSuffix(str, m, n) {
     return text.substring(m, text.length - n);
   }
   ```

2. Subject to the above restrictions, show code for a function `lineAt(text, offset)` which, when given a string `text` and index `offset`, returns the line at index `offset` in string `text`. A line is defined to be a maximal sequence of characters which do not contain a `'\n'` newline character. *4-points*

   ```
   > lineAt('012\nabcd', 0)
   '012'
   > lineAt('012\nabcd', 1)
   '012'
   > lineAt('012\nabcd\n', 5)
   'abcd'
   > lineAt('012\nabcd', 5)
   'abcd'
   > lineAt('012\nabcd', 3)
   ''
   ```

Use `lastIndexOf()` to get the index of the newline preceeding offset; note the addition of 1 to the result of `lastIndexOf()` to point to the start of `text` when we are at the first line which will not have a preceeding newline. Use `indexOf()` to get the index of the newline following `offset`; note the addition of a newline character to `text` to handle the situation where it does not end with a newline. We extract the line using `substring()` and make a final use of `replace()` to handle the situation where `offset` indexes a newline character.

```
function lineAt(text, offset) {
  return text.
    substring(text.lastIndexOf('\n', offset)+1,
              (text + '\n').indexOf('\n', offset)).
    replace('\n', ”);
}
```

3. Subject to the above restrictions, show code for a function `fixedLengthLines(text, len)` which returns `text` with all lines within text with length set to `len`. When a line is shorter than `len` it is padded on the right with the requisite number of spaces; when it's length is greater than `len`, the requisite number of suffix characters are removed. Note that a line is a maximal sequence of characters not containing a newline character `'\n'`.

   All lines in the return value must always be followed by a `'\n'` character irrespective of whether that is the case for the corresponding line in `text`. *4-points*

```
> fixedLengthLines('12345\n1\n12', 3)
'123\n1  \n12 \n'
> fixedLengthLines(”, 3)
'   \n'
```

Use `split()` to get the lines in `text`, `padEnd()` or `substr()` as appropriate to create a new line of the specified `len`, add in a newline to each line, followed by a final `join()` to stick everything back together again into a single string.

```
function fixedLengthLines(text, len) {
  return text.split('\n').
    map(line => (line.length < len)
                ? line.padEnd(len) + '\n'
                : line.substr(0, len) + '\n').
    join(”);
}
```

4. Subject to the above restrictions, show code for a function `oddLengthLines(text)` which, when given a string `text`, returns `text` with all lines which have even length (not counting the `'\n'`) removed. Note that a line is a maximal sequence of characters not containing a newline character `'\n'`.

   All lines in the return value must always be followed by a `'\n'` character irrespective of whether that is the case for the corresponding line in `text`. *4-points*

   ```
   > oddLengthLines('01\n012\n0123\n01234\n')
   '012\n01234\n'
   > oddLengthLines('01\n012\n0123\n01234')
   '012\n01234\n'
   > oddLengthLines('')
   ''
   > oddLengthLines('01')
   ''
   > oddLengthLines('0')
   '0\n'
   ```

   Use `split()` to get the lines in `text`, `filter()` to only select odd-length lines, a `map` to stick on newline terminators, followed by a final `join()` to stick everything back together again into a single string.

   ```
   function oddLengthLines(text) {
     return text.split('\n').
       filter((line) => line.length%2 === 1).
       map((line) => line + '\n').
       join('');
   }
   ```

5. Subject to the above restrictions, show code for a function `positiveEvens(arr)` which, when given an array `arr` of integers, returns an array of those elements in `a` which are even and positive. *3-points*

   ```
   > positiveEvens([5, -4, 0, 2])
   [ 2 ]
   ```

   This is a straight-forward application of `filter()`:

   ```
   function positiveEvens(a) {
   ```

```
          return a.filter((e) => e > 0 && e%2 === 0);
        }
```

6. Subject to the above restrictions, show code for a function `stringsLength¬(strings)` which, when given an array `strings` of strings, returns the sum of the lengths of all the strings in `strings`. *3-points*

```
        > stringsLength(['hello', 'world', ”])
        10
```

   This is a straight-forward application of `reduce()`. We need to accumulate the sum of the `length` property of all the strings in an accumulator initialized to 0.

```
        function stringsLength(strings) {
          return strings.reduce((acc, s) => acc + s.length,
0);
        }
```

7. Subject to the above restrictions, show code for a function `selectIndexes(arr, indexes)` which, when given an array `arr` of arbitrary JavaScript objects and an array `indexes` of non-negative integers, returns an array `selects[]` such that `selects.length === indexes.length` and `selects[i]` is `arr[indexes[i]]`. *3-points*

```
         > selectIndexes(['hello', 42, 'world'], [2, 1, 4])
         [ 'world', 42, undefined ]
```

   Simply `map()` the `indexes` to select the `arr` elements:

```
         function selectIndexes(arr, indexes) {
           return indexes.map((e) => arr[e]);
         }
```

8. Subject to the above restrictions, show code for a function `seq(m, n¬)` which, when given integers `m` and `n` with `m <= n`, returns an array containing the integers from `m` (inclusive) to `n` (exclusive). *3-points*

```
        > seq(4, 5)
        [ 4 ]
        > seq(4, 8)
        [ 4, 5, 6, 7 ]
        > seq(4, 4)
        []
        > seq(-3, 4)
        [ -3, -2, -1, 0, 1, 2, 3 ]
```

Simply return the indexes of a `n - m` element array offset by `m`.

```
        function seq(m, n) {
          return new Array(n - m).fill(0).map((e, i) => i +
m);
        }
```

9. Subject to the above restrictions, show code for a function `positiveIndexes(arr)` which, when given an array `arr` of integers, returns an array of the indexes of those elements in `a` which are positive. *5-points*

```
        > positiveIndexes([5, -4, 0, 2])
        [ 0, 3 ]
```

At first glance this seems to be related to `filter()` but `filter()` does not allow returning indexes; instead, we can use `reduce()` to accumulate only the indexes of positive elements.

```
        function positiveIndexes(a) {
        return a.reduce((acc, e, i) => e > 0 ? acc.concat([i])
: acc,
                          []);
        }
```

10. Subject to the above restrictions, show code for a function `nPermutations(arr)` which, when given an array `arr` of arbitrary JavaScript objects, returns the number of permutations of that array. Note that all array elements in `arr` are always regarded as distinct. *5-points*

```
        > nPermutations([])
        1
        > nPermutations([1, 2, 3])
        6
        > nPermutations([1, 2, 3, 3, 1, 1])
        720
        >
```

Assuming that all elements are treated as distinct means that the function merely needs to compute the factorial of the length of the array. This is trivial to do by reducing the incremented array indexes using a multiplication function.

```
function nPermutations(arr) {
  return new Array(arr.length).fill(0).
    reduce((acc, e, i) => acc*(i + 1), 1);
}
```

11. Subject to the above restrictions, show code for a function `fib(n)` which, when given a positive integer `n > 0`, returns the `n`'th Fibonacci number. *6-points*

```
> fib(1)
1
> fib(2)
1
> fib(3)
2
> fib(6)
8
```

Use `reduce()` with an accumulator which is a pair containing the two previous Fibonacci numbers. For call to the `reduce()` function return an accumulator which contains the next pair. Finally, take care of initial conditions and pulling the final Fibonacci number out of the accumulator:

```
function fib(n) {
  return n <= 2
         ? 1
         : new Array(n - 2).fill(0).
            reduce((acc) => [acc[1], acc[0]+acc[1]],
                          [1, 1])
                         [1];
}
```

12. Subject to the above restrictions, show code for a function `fibValues(n)` which, when given a positive integer `n > 0`, returns a `n`-element array `fibs[]` such that `fibs[i]` is `i`'th Fibonacci number. *7-points*

```
> fibValues(1)
[ 1 ]
> fibValues(2)
[ 1, 1 ]
> fibValues(10)
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Use `reduce()` with an accumulator accumulating the Fibonacci values:

```
function fibValues(n) {
  return n < 2
         ? [1]
         : new Array(n - 2).fill(0).
        reduce((acc, e, i) => acc.concat([acc[i]+acc[i+1]]),
                     [1, 1]);
}
```

13. There is a mistake in the requirements for *Project 1* which makes it difficult for humans and other programs to understand the output of the project. How would you fix the requirements to avoid this problem. *5-points*

    There should be a single empty line printed after each matching document. This makes it easier for both humans and programs to identify the results for each document.

14. When working on *Project 1*, a student decided to experiment with JavaScript Map objects.

    ```
    > m = new Map();
    Map {}
    > m.set('a', 1) //set value
    Map { 'a' => 1 }
    > m.get('a') //get it
    1
    > m['b'] = 2 //try using more convenient [] notation
    2
    > m['b'] //it works!
    2
    ```

    The student first tried the `get()` and `set()` methods as per the documentation, but then found that the more convenient `[]`-indexing operator also worked. So the student decided to do their project using the `[]`-indexing operator with `Map`'s. The project worked perfectly. It turns out that the student was wrong in using the `[]`-indexing operator with `Map`'s, so why did the project still work? *10-points*

    `Map`'s are `Object`'s. Hence the `[]` operations are accessing object properties rather than `Map` keys. As mentioned in the project assignment, JavaScript `Object`'s can also be used as `Map`'s. If the particular JavaScript implementation used by the student retained the insertion order in `Object`'s, then the project would appear to work perfectly even though the `Map` datastructure is not being used at all.

15. Give regex's which precisely describe:

    (a) All binary strings of 4-or-more `0`'s?

    (b) All binary strings of odd length containing alternating `0`'s and `1`'s.

(c) All binary strings over `0` and `1` representing numbers greater than 5 when interpreted as binary numbers.

(d) All binary strings over `0` and `1` representing numbers which are evenly divisible by 4 when interpreted as binary numbers.

(e) All binary strings of length less than or equal to 5 containing only `0`'s and `1`'s where the number of `0`'s is equal to the number of `1`'s. *10-points*

We assume that an empty string is a string of even length.

(a) `/0000+/` or `/0{4,}/`.

(b) `/1(01)*|0(10)*/`.

(c) First considering only 3-bit numbers greater than 5: the least-significant 3 bits must be `110` (representing 6) or `111` (representing 7); hence a regex for these 3 bits would be `/11[01]/`. We also need to handle binary numbers having more significant bits than 3. This can be handled using `/0*1[01]{3,}/`, which allows an arbitrary number of non-significant leading `0`'s, a leading `1` followed by 3 or more arbitrary bits. Hence the required regex would be `/11[01]|0*1[01]{3,}/`.

[Thanks to Mr. Santosh Hegde for pointing out a mistake in the original solution.]

(d) For a binary number to be evenly divisible by 4, it must either be `0` or have its least-significant 2 bits as `00`; hence the regex is `/0|(0|¬1)*00/`.

(e) Since the number of `0`'s must be equal to the number of `1`'s, it must be the case that the length of the regex must be even; i.e. have lengths of 0, 2 or 4. Hence a matching regex is `/|01|10|0011|0101|0110¬|1001|1010/`. Note the empty string indicated by the initial `|` may not be acceptable to some regex-engines; in that case, we could use the alternate regex: `/(01)?|10|0011|0101|0110|1001|1010/`.

Note that it is a result of automata theory that it is impossible to write a regex which matches strings containing the same number $n$ of `a`'s and `b`'s for arbitrary $n$. However, for any specified $n$, it is possible to do so by enumerating all possibilities as in the regex provided above.

16. Give precise but compact descriptions for the strings described by the following regex's. If possible, try to relate the matching strings to the syntax of common programming languages.

(a) `/^[-+]\d+/m`

(b) /0[bB][01_]+/

(c) `/[-+]?(?:\d*\.\d+|\d+\.\d*)$/`

(d) `/\’[^\\\’\n]|\\.\’/`

(e) `/\"(?:[^\\\"\n]|\\.)*\"/` *10-points*

The answers follow:

(a) A signed base-10 integer at the start of a line.

(b) `0` followed by a `b` or `B` followed by one-or-more `0`, `1` or underscores. This syntax is supported by languages like JavaScript, Python and Ruby to allow number literals in binary. Ruby allows `_` within numbers for readability (JavaScript and Python do not).

(c) An optionally signed base-10 decimal number at the end of a string. The number must contain a decimal point and there must be at least one digit before or after the decimal point.

(d) A ’-quoted string containing either a single character (which cannot be `\’,` ’‘ or newline); or a \ followed by any character other than newline. Hence this looks like a character literal in programming languages like Java or C, with \ used for introducing escape sequences.

(e) A `"`-quoted string containing zero-or-more occurrences of characters other than \, `"` or newline or a \ followed by any character other than newline. Hence this looks like a `"`-quoted string in programming languages like Java or C, with \ used for introducing escape sequences.

17. Here is an example of a simple *HTML document*:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <title>Sample</title>
  </head>
  <BODY>
    <h1>A Sample Document</h1>
    <p class="sample-para">
     Some <strong>strong and <em>emphasized</em> text</strong>
    </p>
  </BODY>
</html>
```

The HTML contains markup within `<` and `>` angle-brackets tags. Each tag has a case-insensitive element name; ending tags start with `</`. The actual contents of the document is the content of the `body` element.

Find bugs and inadequacies in the following funtion which purports to extract the plain-text content of a HTML document. Then provide a fixed version of the function.

```
/** Given a string html for a HTML document, return the text content
 *  with all the HTML markup removed.  Specifically, remove header
 *  info up to and including the initial <body> tag and the footer
 *  including </body> and beyond.  Also strip out all remaining
 *  HTML tags as well as empty lines.
 */
function htmlToText(html) {
  return html.
    replace(/(.|\n)*\<body.+\>/, ''). //remove up till body
    replace(/\<\/body(.|\n)*/, ''). //remove from </body
    replace(/\<.+\>/, ''). //remove tags
    replace(/^\s*$/, '');  //remove empty lines
}
```

The extracted text for the sample document should be something like:

```
A Sample Document
  Some strong and emphasized text
```

*15-points*

The problems include the following:

- The most serious problem is the use of the regex /.+/ to match the rest of a tag. Since .+ matches the rest of a line, the regex could skip over other tags on the same line. A better regex to match the rest of a tag would be /[^>]+/.
- The regex's which look for `<body` and `</body` should be case insensitive.
- The intent behind the `$` in the regex for removing empty lines is to indicate the end of a line (this is incorrect without the use of the `/m` flag). However, the `$` merely sets the context for the rest of the regex, it does not actually match a newline. Hence to actually remove empty lines completely, the regex would need to include /\n/.
- The regex for removing empty lines should use the `/m` flag to force `^` to anchor at the start of a line instead of the start of a string.
- Many of the replacements need to be specified as *global* using the `/g` flag.

Fixing these problems results in:

```
function htmlToText(html) {
  return html.
    replace(/(.|\n)*\<body[^\>]*\>/i, ''). //remove header
    replace(/\<\/body(.|\n)*/i, ''). //remove footer
    replace(/\<[^\>]+\>/g, ''). //remove tags
    replace(/^\s*\n/gm, '');  //remove empty lines
}
```