1 Homework 1

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'
   ```

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)
   ''
   ```

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'
```

4. Subject to the above restrictions, show code for a function `oddLength-Lines(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'
```

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 ]
```

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
```

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 ]
```

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 ]
```

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 ]
```

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
>
```

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
```

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 ]
    ```

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*

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*

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*

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*

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