1 Homework 1 Solution

Due Date: Jun 11 by 11:59p;

To be turned in as **PDF** via submission link on mycourses. To facilitate grading, please also add a `hw1.js` executable file for Questions 1 - 13.

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

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.

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

It may be the case that some questions cannot be answered as requested.

Many of the questions are meant to familiarize you with the built-in functions available in JavaScript (and many other languages) for arrays and strings.

Notes:

- We use `C:fn()` to refer to `C.prototype.fn()`.
- A **word** is defined to be a maximal contiguous sequence of non-whitespace characters.

Restrictions

Some of the questions specify "Subject to the above restrictions". These restrictions are to force you to write code in a **strictly** functional style without any mutation. The specific restrictions are:

- Your code may not make any explicit use of destructive assignment, iteration or recursion.
- You code may not contain any `let` or `var` declarations.
- The answer provided for a specific question may contain only a **single** top-level function.

What you are allowed to do:

- Your code may declare **const** variables with an initializer.
- A function provided for a particular answer may call a function defined in an earlier answer.
- You may also use the full power of JavaScript *regular expressions* for functions involving manipulating text.
- You may use any String, RegExp, Array or Number or Math functions which are used only for their return value and not for any side-effects. So for example, you may use Array:reverse() if you are only using its return value and not for its side-effect of changing its argument.

Some hints for writing code subject to the above restrictions:

- In the absence of assignment and iteration, the only sequence of statements you can write are a sequence of zero-or-more `const` declarations followed by a `return` statement, or an `if-then-else` statements with the bodies of the `then` and `else` subject to the same restrictions.
- Instead of using `if-then-else` statements, you are strongly urged to consider using 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 including the current index of the element being operated on and the array being operated on.
- The `Array.from()` function may be useful for setting up initial arrays.
- **Warning**: One of the bad parts of JavaScript is that when `return`'ing a value from a function, the expression representing the returned value must start on the same line as the `return` keyword. So

  ```
  return
    expr;
  ```

  will return `undefined`, but

  ```
  return expr;
  ```

  or

  ```
  return (
    expr
  );
  ```

  will work.

- 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 + 1) for all i < n
   */
  function factValues(n) {
    return Array.from({length: n-1}, (_, i) => i + 2).
      //[2, 3, 4, ..., n]
      reduce((acc, e, i) => acc.concat([e * acc[i]]),
             [1]);  //[1, 1*2, 1*2*3, ...,   1*2*3*...*n]
  }
  ```

  We create an initial array of length `n - 1` with initial values $2 \ldots n$; 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[i]` to pick up the last acc value, we could have used `acc.slice(-1)[0]` instead.

1. Subject to the above restrictions, show code for a function `max(nums)` which when given a non-empty list `nums` of numbers, returns the maximum value in `nums`.

   Hint: Use Array::sort(). *4-points*

   ```
   > max([4, 3, 7, 1])
   7
   > max([4])
   4
   max([-1.2, -0.8, -2.4])
   -0.8
   >
   ```

   Simply return the last element of the sorted array. We use Array.from() to ensure that `sort()`'s changing its argument does not change the parameter to `max()` (not changing parameters was not specified in the restrictions but would definitely be in the spirit of the restrictions).

   ```
   function max(nums) {
     assert(nums.length > 0);
    return Array.from(nums).sort((a, b) => a - b)[nums.length
   - 1];
   }
   ```

2. Subject to the above restrictions, show code for a function `average(nums)` which when given a non-empty list `nums` of numbers, returns the average of the values in `nums`. *4-points*

   ```
   > average( [1, 1, 3, 3, 4, 6 ])
   3
   > average( [1, 1, 2, 3, 4 ])
   2.2
   > average( [11.2])
   11.2
   >
   ```

   Simply use `reduce()` to compute the sum and then compute the average.

   ```
   function average(nums) {
     assert(nums.length > 0);
     const sum = nums.reduce((acc, v) => acc + v);
     return sum/nums.length;
   }
   ```

3. Subject to the above restrictions, show code for a function `linMax(nums)` which when given a non-empty list `nums` of numbers, returns the maximum value in `nums`. The performance of `linMax()` must be $O(n)$ where n is the length of `nums`. *5-points*

```
> linMax([4, 3, 7, 1])
7
> linMax([4])
4
> linMax([-1.2, -0.8, -2.4])
-0.8
>
```

Simply use `reduce()`. Since `nums` is guaranteed to be non-empty, there is no problem initializing the `reduce()` accumulator argument with a **minimum** value.

```
function linMax(nums) {
  assert(nums.length > 0);
  return nums.reduce((acc, e) => e > acc ? e : acc);
}
```

4. Subject to the above restrictions, show code for a function `maxAbs(nums)` which when given a non-empty list `nums` of numbers, returns the maximum absolute value in `nums`. *4-points*

```
> maxAbs([4, 3, 7, 1])
7
> maxAbs([4])
4
> maxAbs([-1.2, -0.8, -2.4])
2.4
>
```

Simply map the list using `Math.abs()`, sort and return last element.

```
function maxAbs(nums) {
  assert(nums.length > 0);
  return nums.
    map(v => Math.abs(v)).
    sort((a, b) => a - b)
    [nums.length - 1];
}
```

5. Subject to the above restrictions, show code for a function `median(nums)` which when given a non-empty list `nums` of numbers, returns the median of `nums`. *5-points*

```
> median([3, 1, 3, 9, 8, 7, 6])
6
> median([8, 6, 9, 2, 3, 4, 1, 5, ])
4.5
> median([8, ])
8
> median([8, 2 ])
5
> median([8, 2, 3])
3
> median([1.4, 2.4])
1.9
>
```

Sort `nums` and then pick up the median as the middle element when `nums` contains an odd number of elements; the average of the two middle elements when `nums` contains an even number of elements.

```
function median(nums) {
  const len = nums.length;
  assert(len > 0);
  const len2 = Math.trunc(len/2);
  const sorted = nums.sort();
  return (len%2 === 1)
         ? sorted[len2]
         : (sorted[len2 - 1] + sorted[len2])/2;
}
```

6. Subject to the above restrictions, show code for a function `runs(ints)` which when given a non-empty list `ints` of integers, returns a list of runs of values `ints`. Each run is represented as a non-empty list of repeated numbers. *6-points*

```
> runs([1, 1, 2, 2, 2, 3])
[ [ 1, 1 ], [ 2, 2, 2 ], [ 3 ] ]
> runs([1,])
[ [ 1 ] ]
> runs([ 2, 2, 2, 3, 3, 1, 1, 3, 3])
[ [ 2, 2, 2 ], [ 3, 3 ], [ 1, 1 ], [ 3, 3 ] ]
> runs([ -2, -2, -2, -3, -1, -3,  -3, -3, ])
[ [ -2, -2, -2 ], [ -3 ], [ -1 ], [ -3, -3, -3 ] ]
>
```

Accumulate runs by comparing the current element with the run being currently accumulated.

```
function runs(ints) {
  assert(ints.length > 0);
  const extendRuns = (runs, val) => {
    const lastRun = runs.slice(-1)[0];
    assert(lastRun.length > 0);
    return (lastRun[0] === val)
        ? runs.slice(0, -1).concat([lastRun.concat([val])])
          : runs.concat([[val]]);
  };
  return ints.slice(1).reduce(extendRuns, [[ints[0]]]);
}
```

7. Subject to the above restrictions, show code for a function `mode(ints)` which when given a non-empty list `ints` of integers, returns the mode of `ints`. If there are multiple modes values having equal counts, then the largest value should be returned. *6-points*

```
> mode([ 2, 2, 2, 3, 3, 1, 1, 3, 3])
3
> mode([ 2, 2, 2, 3, 3, 1, 1, 3, 3, 2])
3
> mode([ 2, 2, 2, 3, 2, 3, 1, 1, 3, 3, 2])
2
> mode([ 2, ])
2
> mode([ -2, -2, -2, -3, -1, -1, -3, -3, -2])
-2
> mode([ -2, -2, -2, -3, -1, -1, -3, -3, -3,  -2])
-2
> mode([ -2, -2, -2, -3, -1, -3, -1, -3, -3, -3,  -2])
-3
```

Sort the list, compute runs and then return element having the longest run.

```
function mode(ints) {
  assert(ints.length > 0);
  const intsRuns = runs(ints.sort((a, b) => a - b));
  const sortedRuns = intsRuns.sort((run1, run2) => {
    const [len1, len2] = [run1.length, run2.length];
```

```
      return len1 === len2 ? run1[0] - run2[0] : len1 - len2;
    });
    return sortedRuns.slice(-1)[0][0];
  }
```

8. Subject to the above restrictions, show code for a function `isDivisible¬ (binStr, d)` which when given a non-empty binary string containing only `0` or `1` characters, returns true iff the binary number specified by `binStr` is divisible by positive integer `d`. *2-points*

   ```
   > isDivisible('1111', 5)
   true
   > isDivisible('1111', 4)
   false
   > isDivisible('1100', 4)
   true
   > isDivisible('1100', 3)
   true
   > isDivisible('111', 1)
   true
   >
   ```

   Trivial after using `Number.parseInt()` to convert from the binary string to a number.

   ```
   function isDivisible(binStr, d) {
     return Number.parseInt(binStr, 2) % d === 0;
   }
   ```

9. Subject to the above restrictions, show code for a function `words(text)` which returns a list of all the words in `text`. For this and subsequent questions, a **word** is defined as a maximal sequence of non-space characters. *2-points*

   ```
   > words('  twas brillig and\n the slithy\n\t toves ')
   [ 'twas', 'brillig', 'and', 'the', 'slithy', 'toves' ]
   > words('  twas ')
   [ 'twas' ]
   > words('  ')
   []
   > words('')
   []
   >
   ```

   Simply use `String:split()` but need to take care of the fact that `split()`

can produce empty strings when splitting an empty string.

```
function words(text) {
  const trimmed = text.trim();
  return trimmed.length > 0 ? trimmed.split(/\s+/) : [];
}
```

10. Subject to the above restrictions, show code for a function `revWords¬(text)` which returns string containing all the words in text in reverse order. The exact spelling of whitespace in the return value must match the spelling of the corresponding whitespace in `text`. *6-points*

```
> revWords(
   '  twas brillig   and\n the slithy\n\t toves   ')
'  toves slithy   the\n and brillig\n\t twas   '
> revWords('  ')
'  '
> revWords('   twas ')
'   twas '
> revWords('twas')
'twas'
> revWords(”)
”
>
```

Compute array of reversed words and array of whitespace and then merge them together, being careful to ensure that whitespace segments are inserted between the correct words. Finally `join()` array together.

```
function revWords(text) {
  const trimmed = text.trim();
  const rev =
    (trimmed.length > 0 ? trimmed.split(/\s+/) : []).
    reverse();
  const [ leadSpace ] = text.match(/^\s*/);
  const [ trailSpace ] =
    trimmed.length > 0 ? text.match(/\s*$/) : ”;
  const interSpaces = text.trim().
    split(/\S+/).
    filter(space => space.length > 0);
  return [leadSpace].
    concat(rev.slice(0, -1).
           flatMap((w, i) => [w, interSpaces[i]])).
    concat(rev.slice(-1)).
    concat([trailSpace]).
    join('');
}
```

11. Subject to the above restrictions, show code for a function `wordCounts(text)` which returns an object mapping each distinct word in text to a count of the number of times it occurs within text. The keys of the return value should be sorted in lexicographical order. Words which differ in case should be treated as distinct. You may assume that no word looks like an integer.

    Hint: Since ES6, the non-integer keys of an object are guaranteed to be sorted in insertion order. *5-points*

    ```
    wordCounts(' aaa bb aa bb aaa Aaa')
    { Aaa: 1, aa: 1, aaa: 2, bb: 2 }
    > wordCounts(' aaa bb aa\n\tbb aaa Aaa')
    { Aaa: 1, aa: 1, aaa: 2, bb: 2 }
    > wordCounts(' aaa, bb, aa,\n\tbb, aaa, Aaa')
    { Aaa: 1, 'aa,': 1, 'aaa,': 2, 'bb,': 2 }
    > wordCounts('x')
    { x: 1 }
    > wordCounts('')
    {}
    > wordCounts('  ')
    {}
    >
    ```

    Compute runs of sorted words and then use the runs to compute the word-count object.

    ```
    function wordCounts(text) {
      const sorted = words(text).sort();
      const wordRuns = sorted.length > 0 ? runs(sorted) : [];
      return Object.fromEntries(
        wordRuns.map(run => [run[0], run.length])
      );
    }
    ```

12. Subject to the above restrictions, show code for a function `modeWord(text)` which returns the word which occurs most often in `text` (which is guaranteed to contain at least one word). If multiple distinct words occur most often, then the returned word should be the lexicographical smaller word. *5-points*

    ```
    > modeWord(' aaa bb aa\n\tbb aaa Aaa')
    ```

```
'aaa'
> modeWord(' aaa bb aa\n\tbb aaa Aaa bb')
'bb'
> modeWord(' x ')
'x'
> modeWord('x')
'x'
>
```

Pick up word with maximum word-count.

```
function modeWord(text) {
  return Object.entries(wordCounts(text)).
    sort((a, b) => a[1] !== b[1] ? b[1] - a[1]
                                 : (a[0] < b[0] ? -1 : +1))
    [0][0];
}
```

13. Subject to the above restrictions, show code for a function `primes(n)` which returns all the prime numbers less than or equal to positive integer `n`. *6-points*

```
> primes(10)
[ 2, 3, 5, 7 ]
> primes(11)
[ 2, 3, 5, 7, 11 ]
> primes(100)
[
   2,  3,  5,  7, 11, 13, 17, 19,
  23, 29, 31, 37, 41, 43, 47, 53,
  59, 61, 67, 71, 73, 79, 83, 89,
  97
]
> primes(1)
[]
>
```

Candidates for primes are from $2 \ldots n$. Initializing primes to the first candidate, for each remaining candidate add it to primes if it is not a multiple of a previously computed prime.

```
function primes(n) {
  if (n < 2) {
    return [];
  }
  else {
    const candidates = Array.from({length: n - 2 + 1}).
      map((_, i) => i + 2);
    return candidates.
      reduce((acc, i) =>
                acc.some(d => i%d === 0)
                ? acc
                : acc.concat([i]),
              [candidates[0]]);
  }
}
```

14. Given a vocabulary Σ containing 2 symbols `a` and `b`, give regex's for the following: *10-points*

    (a) All strings over Σ which start with `b` and end with `a`.

    (b) All strings over Σ which have length less than 10 and end with `a`.

    (c) All strings over Σ whose length is divisible by 10.

    (d) All strings over Σ which contain more `b`'s than `a`'s.

    (e) All strings over Σ which contain 10 or more `a`'s.

    The answers follow:

    (a) `b(a|b)*a`. String consists of a `b` follows by any possibly empty string over Σ, followed by a single `a`.

    (b) `(a|b){0,8}a`. String consists of any possibly empty string over Σ containing up to 8 characters followed by a single `a`.

    (c) `((a|b){10})*`. Zero-or-more repetitions of strings over Σ having length 10.

    (d) Cannot be done since regex's cannot count.

    (e) `(b*ab*){10,}`. 10-or-more `a`'s with each `a` possibly preceeded / followed by 0-or-more `b`'s.

15. Traditionally, JavaScript used objects to implement the functionality of a dictionary which maintains a mapping between keys and values. Recently, JavaScript added the Map datatype.

    Compare the use of JavaScript objects and `Map`'s to implement dictionaries. *15-points*

    Oops, I did not realize that MDN has added the comparison directly in their Map reference. Anyway, the answer I had in mind is along the lines of the comparison provided by MDN:

- Object keys are restricted to `String`'s (also `Symbol`'s which we have not yet covered). The keys to a `Map` can be any JavaScript value.
- Objects come with a prototype which may have other keys. With maps the only keys are those which are added.
- Since ES6, objects preserve insertion order among `String` keys. Maps preserve insertion order among **all** keys.

  ```
  > x = {}
  {}
  > x.a = 22
  22
  > x[1] = 33
  33
  > x
  { '1': 33, a: 22 }
  > y = new Map()
  Map(0) {}
  > y.set('a', 22)
  Map(1) { 'a' => 22 }
  > y.set('1', 33)
  Map(2) { 'a' => 22, '1' => 33 }
  >
  ```

- It is possible to iterate over a `Map` directly. Given the above `x` and `y`:

  ```
  > for (const x of y) console.log(x);
  [ 'a', 22 ]
  [ '1', 33 ]
  undefined
  >
  ```

  Iterating over an object needs to be done manually. Something like:

  ```
  for (const pair of Object.entries(x)) console.log(pair);
    [ '1', 33 ]
    [ 'a', 22 ]
    undefined
    >
  ```

  which is actually iterating over an array.

- Obtaining the number of elements of a `Map` can be done easily using its `size()` method. For an object, it must be done manually often by building up an array (using the `keys()`, `entries()` or `values()`

methods from `Object`) and getting its `length`.

- It is likely that `Map`'s are implemented more efficiently than objects.

16. Discuss the validity of the following statements. What is more important than whether you ultimately classify the statement as **true** or **false** is your justification for arriving at your conclusion. *15-points*

    (a) The binary relation `<` is trichotomous on JavaScript numbers.

    (b) Given a JavaScript declaration `const x = {};`, then the assignment `x.a = 22;` is illegal as it modifies `x`.

    (c) If `s` is a non-empty string, then the assignment `s[0] = 'a';` will cause an error since JavaScript strings are immutable.

    (d) It is possible to write a JavaScript string literal so that it spans multiple lines.

    (e) If `string.match(regex)` succeeds with value `m`, then `m.length === n + 1` where `n` is the number of parentheses pairs in `regex`.

    The answers follow:

    (a) JavaScript numbers are not trichotomous because of the presence of `NaN`. It will be the case that all of $a < b$, $a = b$ and $a > b$ are **false** if either a or b are `NaN`. Hence the statement is **false**.

    (b) The declaration declares `x` as a constant reference to an object; i.e. the object itself is not declared constant. Assigning directly to `x` is illegal, but assigning to the properties of the object referenced by `x` is permissible and the statement is **false**.

        [It is possible to make the properties of `x` immutable using `Object¬ .freeze(x)`.]

    (c) Strings are indeed immutable and the assignment will fail, but silently without any error. Hence the statement is **false**.

    (d) Template string literals delimited using backquotes ` can span multiple lines. Hence the statement is **true**.

    (e) If `string.match(regex) === a`, then `a.length === n + 1` where `n` is the number of **capturing** parentheses pairs in `regex`. Since some of the parentheses pairs in `regex` may not be capturing, the statement is **false**.