Week 01

Seth Childers

1 Exercise 01

Problem: Design an algorithm for computing n for any positive integer n. Besides assignment and comparison, your algorithm may only use the four basic arithmetical operations.

Code written in JavaScript

```
/**
 * Exercises 1.1 - #4
 * Design an algorithm for computing the floored square root for any positive
 * integer n. Besides assignment and comparison, your algorithm may only use
 * the four basic arithmetical operations.
 *
 * @param {number} number
 */
function flooredSquareRoot(number) {
    let i = 1;
    let result = 1;
    while (result <= number) {
        i++;
        result = i * i;
    }
    return i - 1;
}

// put user input here
let root = flooredSquareRoot(141);
console.log(root);
```

Results

```
141 should be 11: 11
111 should be 10: 10
15 should be 3: 3
19 should be 4: 4
54 should be 7: 7
254 should be 15: 15
369 should be 19: 19
```

2 EXERCISE 02

Problem: What does EuclidâĂŹs algorithm do for a pair of integers in which the first is smaller than the second? What is the maximum number of times this can happen during the algorithmâĂŹs execution on such an input?

Code written in JavaScript

```
/**
 * Exercises 1.1 - #8
 * What does Euclid's algorithm do for a pair of integers in which the first is
 * smaller than the second?
 *
 * Euclid's algorithm will switch the two numbers in the first iteratation, but
 * it will still solve for the GCD correctly, as if the first number were
 * larger.
 *
 * What is the maximum number of times this can happen
 * during the algorithm's execution on such an input?
 *
 * This will only happen for the first iteration of the Euclidean algorithm.
 * After this first iteration, the numbers will switch and the first number
 * will then be the larger number and the second number will then be the
 * smaller. This switch will only happen on the first iteration if the first
 * number is smaller.
 *
 * @param {number} m Non-zero integer
 * @param {number} n Non-zero integer
 */
function findGCD(m, n) {
    let i = 0;
    let remainder = 1;

    while (n !== 0) {
        i++;
        remainder = m % n;
        m = n;
        n = remainder;
```

```
    }

    return {
        gcd: m,
        loopCount: i
    };
}

// put user input here
let result = findGCD(10, 12);
console.log(`findGCD(10, 12)`);
console.log(`The GCD is: ${result.gcd}`);
console.log(`The number of times the while loop ran: ${result.loopCount}`);
```

Results

```
findGCD(10, 12)
The GCD is: 2The number of times the while loop ran: 3
```

3 EXERCISE 03

Problem: Write pseudocode for an algorithm for finding real roots of equation ax$\hat{2}$ + bx + c = 0 for arbitrary real coefficients a, b, and c. (You may assume the availability of the square root function sqrt(x).)

Code written in JavaScript

```
/**
 * Exercises 1.2 - #4
 * Write pseudocode for an algorithm for finding real roots of equation
 * ax^2 + bx + c = 0 for arbitrary real coefficients a, b, and c. (You may
 * assume the availability of the square root function sqrt(x).)
 *
 * @param {number} a The value for variable 'a' in the quadratic formula
 * @param {number} b The value for variable 'b' in the quadratic formula
 * @param {number} c The value for variable 'c' in the quadratic formula
 */
function findDiscriminant(a, b, c) {
    let x = 0;
    let discriminant = Math.pow(b, 2) - (4 * a * c);
    let numSolutions = 0;
    if (discriminant < 0) {
        numSolutions = -2;
    } else if (discriminant === 0) {
        numSolutions = 1;
    } else {
```

```
        numSolutions = 2;
    }
    return {
        discriminant,
        numSolutions
    };
}

function calculateRoot(a, b, c, discriminant) {
    let x = -b;
    let roots = [];
    // if the discriminant was zero
    if (discriminant.numSolutions === 1) {
        x = x / (2 * a);
        roots.push(x);
        return roots;
        // if the discriminant was positive
    } else if (discriminant.numSolutions === 2) {
        let x1 = x;
        x = (x - Math.sqrt(discriminant.discriminant)) / (2 * a);
        x1 = (x1 + Math.sqrt(discriminant.discriminant)) / (2 * a);
        roots.push(x);
        roots.push(x1);
        return roots;
        // if the discriminant was negative
    } else if (discriminant.numSolutions === -2) {
        console.log('There are no real roots, but we\'ll log -1 anyways')
        roots.push(-1);
        return roots;
    }
}

function main(a, b, c) {
    // put user input here
    let discriminant = findDiscriminant(a, b, c);
    let roots = calculateRoot(a, b, c, discriminant);

    console.log(`The ${discriminant.numSolutions > 1 ? 'roots': 'root'} for
        ${a}x^2 + ${b}x + ${c} ${discriminant.numSolutions > 1 ? 'are': 'is'}:
        ${roots}`);
    return;
}

main(9, 12, 4);
main(8, 10, 3);
main(15, 2, 1);
```

Results

```
The root for 9x^2 + 12x + 4 is: -0.6666666666666666
The roots for 8x^2 + 10x + 3 are: -0.75,-0.5There are no real roots, but we'll
    log -1 anywaysThe root for 15x^2 + 2x + 1 is: -1
```

4 EXERCISE 04

Problem: Name the algorithms for the searching problem that you already know. Give a good succinct description of each algorithm in English. If you know no such algorithms, use this opportunity to design one.

Code written in JavaScript

```
/**
 * Exercises 1.3 - #2
 * Name the algorithms for the searching problem that you already
 * know. Give a good succinct description of each algorithm in English.
 * If you know no such algorithms, use this opportunity to design one.
 *
 * Algorithm: Binary Search
 * Description: A tree search through a sorted set of elements that starts in
 * the middle of the set, and uses a comparison between the desired
 * element and the current middle element, and moves on to either the lower
 * half or the higher half of the ordered set, depending on the result of the
 * comparison against the middle element and the desired element.
 *
 * Algorithm: Linear Search
 * Description: Start at the beginning or the end of an array. Compare
 * the desired element with the current element. If they match, return
 * the current index of the array. If they do not match, move to the next
 * element in the array by incrementing the index by one.
 *
 * Algorithm: Exponential Search
 * Description: Split the sorted array into exponentially sized groups,
 * starting with size 1, then size 2, 4, 8, etc. As you go and create these
 * sub-arrays, compare their last element with the desired element. If the
 * value is greater than the desired element, then use that sub-array. At this
 * point, perform a binary search on this sub-array.
 */
```

5 EXERCISE 05

Problem: If you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answers for [a] lists represented as arrays. [b] lists represented as linked lists.

Code written in JavaScript

```
/**
 * Exercises 1.4 - #2
 * If you have to solve the searching problem for a list of n numbers,
 * how can you take advantage of the fact that the list is known to be
 * sorted? Give separate answers for
 * a. lists represented as arrays.
 * b. lists represented as linked lists.
 *
 * Arrays: If you know if an array is sorted or not, you can determine which
 * search algorithm would be the most efficient based on the array length. Some
 * algorithms require the list to be sorted, some do not. It will also allow
 * you to determine if sorting it would be more efficient or not, in order to
 * reduce the number of memory accesses.
 *
 * Linked Lists: Knowing if a linked list is sorted or not does not help,
 * seeing as linked lists must be searched linearly, or rather in order one by
 * one. But if it is a doubly linked list, then you would be able to determine
 * which end to start searching from.
 */
```

6 Exercise 06

Problem: For each of the following applications, indicate the most appropriate data structure: Answering telephone calls in the order of their known priorities, and sending backlog orders to customers in the order they have been received.

Code written in JavaScript

```
/**
 * Exercises 1.4 - #9
 * For each of the following applications, indicate the most appropriate
 * data structure:
 *
 * Question: Answering telephone calls in the order of their known priorities
 * Answer: A priority queue, which would sort the items in terms of
 * priority, then you would access them as a sort of modified queue which is
 * popping out the highest priority first.
 *
 * Question: Sending backlog orders to customers in the order they have been
 * received
 * Answer: A queue would fit best here, acting as a FIFO data structure.
 *
 * Question: Implementing a calculator for computing simple arithmetical
 * expressions
 * Answer: A priority queue again, seeing as you need to implement and
```

```
 * prioritize the order of operations in the equation, then after ordering
 * them you can pop each element out in order of priority, or rather in their
 * proper order of operations order
 */
```

7 EXERCISE 07

Problem: There are n lockers in a hallway, numbered sequentially from 1 to n. Initially, all the locker doors are closed. You make n passes by the lockers, each time starting with locker #1. On the ith pass, i =1, 2, ..., n, you toggle the door of every ith locker: if the door is closed, you open it; if it is open, you close it. After the last pass, which locker doors are open and which are closed? How many of them are open?

Code written in JavaScript

```
/**
 * Exercises 1.1 - #12
 * There are n lockers in a hallway, numbered sequentially from 1 to n.
 * Initially, all the locker doors are closed. You make n passes by the
 * lockers, each time starting with locker #1.
 * On the ith pass, i =1, 2, ..., n, you toggle the door of every ith locker:
 * if the door is closed, you open it; if it is open, you close it.
 * After the last pass, which locker doors are open and which are closed? How
 * many of them are open?
 */
function toggleLockers(numLockers) {
    // create an array of length 'numLockers' and fill it with zeros
    let lockers = Array(numLockers).fill(0);
    // create an array with each locker's 1-based index
    let lockerNumbers = lockers.map((locker, i) => i + 1);
    // create an array with the locker's that are perfect squares
    let openLockers = lockerNumbers.filter(locker => Math.sqrt(locker) % 1 ===
        0);
    console.log(`The lockers that are open are: ${openLockers}`);
    return;
}

// put the number of lockers here
let numLockers = 100;
toggleLockers(numLockers);
```

Results

```
The lockers that are open are: 1,4,9,16,25,36,49,64,81,100
```

8 EXERCISE 08

Problem: Create Three Different Algorithms to Solve this Problem. Given two positive numbers A and B, where A is greater than B, nd a way to break up A into B unequal pieces.
For example, if A = 34 and B = 4, then four unequal pieces of A are 6, 7, 9 and 12. These are unequal because there are no duplicate numbers. They break up (or sum up to) 34 because 6 + 7 + 9 + 12 = 34. The numbers representing the pieces (e.g., 6, 7, 9 and 12) must be positive integers (1, 2, 3, etc.), which excludes zero. Note that some pairs of numbers don't work, e.g., 5 and 3, so be sure to error-check that case.

Code written in JavaScript

```
/**
 * Exercises 4.3
 * Create Three Different Algorithms to Solve this Problem. Given two
 * positive numbers A and B, where A is greater than B, and a way to break up A
 * into B unequal pieces.
 *
 * For example, if A = 34 and B = 4, then four unequal pieces of A are 6, 7, 9
 * and 12. These are unequal because there are no duplicate numbers. They break
 * up (or sum up to) 34 because 6 + 7 + 9 + 12 = 34. The numbers representing
 * the pieces (e.g., 6, 7, 9 and 12) must be positive integers (1, 2, 3,
 * etc.), which excludes zero. Note that some pairs of numbers don't work,
 * e.g., 5 and 3, so be sure to
 * error-check that case.
 *
 * Algorithm One:
 * 1. Create array of size 'b' - 1 with each element being its own index
 * 2. Create a variable 'total' that is all the elements in the array added
 * together
 * 3. Subtract the 'total' from 'a' for the last value
 * 4. Each element in the array is a unique number, and the last value is the
 * calculated total from step 3
 *
 * Algorithm Two:
 * 1. Create an array of size b
 * 2. Use a random function to generate a random number between 0 and
 * 'a' - 'b' for each element in the array
 * 3. If each element isn't unique and if the total of the random numbers
 * doesn't add up to 'a', repeat
 * 4. Repeat step 3 as many times as necessary
 *
 * Algorithm Three:
 * 1. Split 'a' into 'b - 1' even groups in an array
 * 2. In a loop, subtract 'i' from a group each iteration (i will increase by
 * one for each element)
 * 3. Iterate i one extra time and save it as the last element in the array
 */
```

```
function algorithmOne(a, b) {
    // create an array of length 'b - 1' and fill it with zeros
    let numArray = Array(b - 1).fill(0);
    // put each elements index as its value in the array, then add the indexes
        up
    let total = numArray.reduce((acc, num, i) => acc += i + 1, 0);
    // subtract the total of your indexes from a to give you the last needed
        value
    let lastNum = a - total;
    // add the last value to the array and return it
    return numArray.concat(lastNum);
}

console.log(algorithmOne(34, 4));
```

Results

```
[ 0, 0, 0, 28 ]
```