WEEK 01

1. Readings

- Levitin Chapter 1

2. Preparation for Assignment

If, and *only if* you can truthfully assert the truthfulness of each statement below are you ready to start the assignment.

2.1. Reading Comprehension Self-Check.

- I know what criterion most classic algorithms satisfy.
- I know what systematically interrupts the narrative flow of the textbook.
- I know to be on the lookout for exercises versus problems, because the chapter exercises in the textbook are not marked with a difficulty level.
- I know where the textbook provides hints to all the exercises.
- I know the properties of logarithms.
- I know the important summation formulas.

2.2. Memory Self-Check.

2.2.1. *Determine Correct Order.* The steps for the best known algorithm for creating algorithms are listed out of order here. What order should they be in?

(1) Decide on: computational means, exact vs approximate solving, data structure(s), algorithm design technique.
(2) Design an algorithm.
(3) Understand the problem.
(4) Prove correctness of the algorithm.
(5) Analyze the algorithm.
(6) Code the algorithm.

2.2.2. *Write a short answer.* Levitin states that one of these problem types is the most difficult to solve. Which is it and why is so difficult to solve?

(1) Sorting
(2) Searching
(3) String Processing
(4) Graph problems
(5) Combinatorial problems
(6) Geometric problems

Date: September 17, 2018.

(7) Numerical problems

3. Week 01 Exercises

3.1. Exercise 4 on Page 7.

3.2. Exercise 8 on page 8.

3.3. Exercise 4 on Page 17.

Write code instead of psuedocode.

3.4. Exercise 2 on page 23.

3.5. Exercise 2 on page 37.

3.6. Exercise 9 on page 38.

4. Week 01 Problems

4.1. Exercise 12 on page 8.

4.1.1. *How might this elisp code help in answering the questions posed in exercise 12?*

```
(require 'cl) ;; for loop

 (defvar doors (make-bool-vector 101 nil))

 (defun flip-doors (n)
   (loop for i from 0 below (length doors)
         when (zerop (mod i n))
         do (aset doors i (if (aref doors i) nil t))))

 (defun flip-100 ()
   (loop for i from 1 to 100 do (flip-doors i))
   (substring (mapconcat (lambda (x) (if x "1" "0")) doors "") 1))
```

4.1.2. *Same algorithm in Swift.*

```
var doors = Array(repeating: 0, count: 101)
func flip_doors(n:Int){
    for i in 0..<doors.count{
        if i % n == 0 {
            doors[i] = abs(doors[i]-1)
        }
    }
}
for i in 1..<doors.count {
    flip_doors(n: i)
}
print(doors.compactMap{String($0)}.joined())
```

4.2. **Exercise 9 on page 25.**

4.3. **Create Three Different Algorithms to Solve this Problem.** Given two positive numbers A and B, where A is greater than B, find 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.

4.4. **Compare/Contrast Your Three Algorithms.** In a similar manner to how Levitin compared and contrasted three different GCD algorithms, evaluate your three algorithms using three different criteria.