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.

Answer:

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

Date: September 16, 2022.

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
(7) Numerical problems

Answer: Combinatorial probelms are the most difficult to solve because they tend to become too big to compute. They grow very fast. We have ways to approximate but it is difficult to find an exact answer.

3. Week 01 Exercises

3.1. **Exercise 4 on Page 7.**

```
for $i \leftarrow 0$ to $n$ do
    if $i \times i > n$ then
        return $i - 1 \times i - 1$
    end
end
```

3.2. **Exercise 8 on page 8.**

Euclids algorithm swaps the m and n if m is less than n. This swap only occurs once in the algorithm.

3.3. **Exercise 4 on Page 17.** Write Clojure code instead of the pseudocode asked for.

3.4. **Exercise 2 on page 23.**

3.4.1. *Binary Search.* This is where you are given a sorted array of numbers and are searching for a single constant. You begin the search at the middle of the array. You then compare the target (number you are searching for) to the current number you are at (the middle). If the target is greater than your current position then jump half way from where you are to the top of the array. If they target is less than your current position then jump half way from your current position to the bottom of the array. The previous position becomes the bottom of the array if you jumped up or the top if you jumped down. Repeat the steps until you converge upon your target. It is possible that your target does not exist in the array.

3.4.2. *Linear Search.* This where you are given either a sorted or unsorted array of constants and are searching for a single or group of constants. You begin at the beginning of the array and then move across the array one by one comparing the current item to what your searching for.

3.5. **Exercise 2 on page 37.**

3.5.1. *Sorted Arrays.* You can take advantage of algorithms like a binary search or linear search depending on how much information you already know about the array. Both have their advantages. Such as, if you know that the target is near the front of the array, a linear search might be better depending on the size of the array. Binary search however has a famed run time.

3.5.2. *Sorted Linked Arrays.* They should not affect it to much since you must start from the head of the linked array and traverse it to the tail to find your item. Linked arrays have better performance in operations like insertion and deletion, however, arrays have better accessing capabilities.

3.6. **Exercise 9 on page 38.**

3.6.1. *Prioriety Queue.*

3.6.2. *Queue.*

3.6.3. *Stack.*

4. Week 01 Problems

4.1. **Exercise 9 on page 25.**

```cpp
#include <iostream>
#include <tuple>
#include <vector>
typedef std::tuple<double, double> point;
typedef std::vector<point> points;

struct Fraction{
    double numerator;
    double denominator;
};

bool intersectingLines(point &A, point &B, point &C, point &D)
{
    // Line AB represented as a1x + b1y = c1
    double a1 = std::get<1>(B) - std::get<1>(A);
    double b1 = std::get<0>(A) - std::get<0>(B);
    double c1 = a1*(std::get<0>(A)) + b1*(std::get<1>(A));

    // Line CD represented as a2x + b2y = c2
    double a2 = std::get<1>(D) - std::get<1>(C);
    double b2 = std::get<0>(C) - std::get<0>(D);
    double c2 = a2*(std::get<0>(C))+ b2*(std::get<1>(C));

    double determinant = a1*b2 - a2*b1;

    if (determinant == 0){
        // The lines are parallel. This is simplified
        // by returning a pair of FLT_MAX
        return false;
    }
    // double x = (b2*c1 - b1*c2)/determinant;
    // double y = (a1*c2 - a2*c1)/determinant;
    return true;

}

// This function determines if the points lie on the same cirumference.
bool circumferencePoints(points &circlePoints){
    /*
    *  Step 01: Determine the mid point between points as they appear in the vector DONE
    *  Mid point (x,y): ((x1+x2) / 2, (y1+y2) /2)
    *  Step 02: Determine gradient between points as they appear in the vector      DONE
    *  Slope y/x: (y2-y1) / (x2 - x1)
    *  Step 03: Find the perpendicular gradient                IN PROGRESS
    *  Perp: Find the recipricol and mulitiply by -1
    *  Step 04: Find additional point for perpendicular point
    *  Step 05: Determine if adjacent pairs of points intersect
    */

    std::cout << "--- Algorithm Starting ---\n";
    points midpoints;
    points intersectPoints;
    std::vector<double> m;
    std::vector<Fraction> slopes;

    // Step 01
    for(auto it = circlePoints.begin(); it != circlePoints.end(); it++){
        double x1 = std::get<0>(*it);
        double y1 = std::get<1>(*it);
        it++;
        double x2 = std::get<0>(*it);
        double y2 = std::get<1>(*it);
        it--;

        point midpoint = point((x1 + x2) / 2, (y1 + y2) /2);
        midpoints.push_back(midpoint);
    }
    // Step 02
    for(auto it = midpoints.begin(); it != midpoints.end(); it++){
        double x1 = std::get<0>(*it);
        double y1 = std::get<1>(*it);
        it++;
        double x2 = std::get<0>(*it);
        double y2 = std::get<1>(*it);
        it--;

        Fraction fraction;
        fraction.numerator = (y2-y1);
        fraction.denominator = (x2 - x1);

        slopes.push_back(fraction);
    }
    // Step 03
    for (auto it = slopes.begin(); it != slopes.end(); it++){
        m.push_back(-1 * (it->denominator / it->numerator));
    }
    // Step 04
    auto itmp = midpoints.begin();
    auto itm = m.begin();

    while(itmp != midpoints.end() || itm != m.end()){
        double b = (*itm * (-1 * std::get<0>(*itmp))) + std::get<1>(*
    itmp);
        double x = 10;
        double y = (*itm * x) + b;
        intersectPoints.push_back(point(x,y));
        ++itmp;
        ++itm;
    }
    // Step 05
    itmp = midpoints.begin();
    auto itin = intersectPoints.begin();
```

```cpp
    int size = midpoints.size();
    int i = 0;

    while(i < size - 1){
        point A = *itmp;
        point B = *itin;
        ++itmp;
        ++itin;
        point C = *itmp;
        point D = *itin;
        --itmp;
        --itin;
        if(intersectingLines(A, B, C, D)){
        }else{
            std::cout << "--- Algorithm Ending ---\n";
            return false;
        }
        ++i;
        ++itmp;
        ++itin;
    }
    std::cout << "--- Algorithm Ending ---\n";

    return true;
}

int main(){
    std::cout << "--- Program Starting ---\n";

    points circlePoints;
    circlePoints.push_back(std::tuple<double, double>(-1.2,1.6));
    circlePoints.push_back(std::tuple<double, double>(-1.85, 0.86));
    circlePoints.push_back(std::tuple<double, double>(1.41, 1.418));
    circlePoints.push_back(std::tuple<double, double>(0.2, 1.99));
    circlePoints.push_back(std::tuple<double, double>(-1.968, 0.356));

    for(auto i = circlePoints.begin(); i != circlePoints.end(); i++){
        std::cout << std::get<0>(*i) << ", ";
        std::cout << std::get<1>(*i) << "\n ";
    }

    if(circumferencePoints(circlePoints)){
        std::cout << "The points form a circle.\n";
    }else{
            std::cout << "The points do not form a circle\n";
        }

        std::cout << "--- Program Ending ---\n";

        return 0;
}
```

4.2. **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.

4.3. **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.

4.3.1. *Comparing/Contrasting.* Algorithm 01 is by far my favorite because the logic of it is intuitive and it runs in $\Omega(n)$ time. The algorithm solves it how it needs to without any major complications. Algorithm 02 is good because it also runs in $\Omega(n)$, however, because the logic is a little more complicated to understand, I think algorithm 01 still holds out as the better algorithm. Algorithm 03 is the worst, easily. It has poor performance and it is overly complicated.

```
#include <iostream>
#include <math.h>

int algorithm01(int A, int B){
    std::cout << "--- Algorithm 01 Starting ---\n";
    // Initialize array
    int nums[B-1];

```

```cpp
    // Break it down
    for(int i = 0; i <= B - 1; i++){
        nums[i] = B - i;
        A -= B - i;
        if(i == B - 1){
            nums[i] = A + 1;
        }
    }
    // Visually inspect
    for(int i = 0; i <= B - 1; i++){
        std::cout << nums[i];
        std::cout << "\n";
    }
    std::cout << "--- Algorithm 01 Ending ---\n\n";
    return 0;
}



int algorithm02(int A, int B){
    std::cout << "--- Algorithm 02 Starting ---\n";
    // DISCLAIMER: I used stack overflow to learn about this greedy algorithm
    int result[B-1];
    int left = A - B*(B + 1)/2;

    for(int i = 0; i < B; i++){
        result[i] = i + 1 + left/B;
        if(i >= B - (left % B)){//Add extra one for last left % n elements
            result[i]++;
    }}
    // Visually inspect
    for(int i = 0; i <= B - 1; i++){
        std::cout << result[i];
        std::cout << "\n";
    }

    std::cout << "--- Algorithm 02 Ending ---\n\n";
    return 0;
}

int algorithm03(int A, int B){
    std::cout << "--- Algorithm 03 Starting ---\n";
    int list[A];
    int sol[B];
    int sum = 0;

    for(int i = 0; i <= A; i++){
        list[i]  = i + 1;
    }
    for(int i = 0; i < A; i++){
        sum = 0;

        for(int j = 0; j < B; j++){
            sol[j] = i + j;
        }
        for(int k = 0; k < B; k++){
            sum += sol[k];
        }
        if(sum == A){
            std::cout << "broke\n";
            break;
        }
        else if(sum > A){
            for(int j = 0; j < B; j++){
                sol[j] = (i) + (j) - 1;
        }
        break;
        }

    }

    sum = 0;
    for(int i = 0; i < B; i++){
        sum += sol[i];
    }
    // Add remainder
    int remainder = A - sum;
    sol[B-1] += remainder;

    // Visually inspect
    for(int i = 0; i < B; i++){
        std::cout << sol[i];
        std::cout << "\n";
    }

    std::cout << "--- Algorithm 03 Ending ---\n\n";

    return 0;
```

```cpp
}

int main(){
    std::cout << "--- Program Starting ---\n";

    int A = 80;
    int B = 6;

    algorithm01(A, B);
    algorithm02(A, B);
    algorithm03(A, B);

    return 0;
}
```