Chapter 1 - Introduction

- An algorithm is a sequence of **unambiguous** instructions for solving a problem (obtaining output for any legitimate input in a ***finite*** *amount of time*).

  > non-ambiguous
  > range of input has to be specified
  > same algorithm can be represented in many ways
  > there could exist several algorithm for a given proble,
  > different algorithms could solve a problem in dramatically different speeds

- Relationship among algorithms, data structures, and programming languages:
  > Algorithms: steps & planning
  > Data Structures: ingredients to use
  > Programming Languages: actual implementation

  Analogy with building a house:
  > Alg: blue print; design efficiency (space, energy, etc.)
  > DS: bricks, doors, frames
  > PL: building the house; you can use different brands of products (different languages)

- Example: Compute the greatest common divisor *gcd(m, n)*
  - Solution 1: Euclid's algorithm `gcd(m, n) = gcd(n, m mod n)`

    Algorithm *Euclid(m,n)*

    ```
    while n != 0:
        r = m % n
        m = n
        n = r
    return m
    ```

    How do we know this algorithm eventually stops?
    The second int of the pair (m mod n) gets smaller with each iteration and eventually becomes 0. Even if m end up being 1, all numbers mod 1 is 0, and the algorithm stops.

  - Solution 2: Consecutive integer checking algorithm: finding the largest integer that divides both

    Algorithm *gcd(m,n)*

    ```
    t = min(m,n)
    while t > 0:
        if m%t == 0 and n%t == 0:
            return t
        t -= 1
    return -1  # when m or n == 0
    ```

  - Solution 3: Middle-school procedure
    > find prime factors of m
    > find prime factors of n
    > identify all common factors
    > compute the product

  - Performance Analysis
    - Solution 1 is the most efficient, as we get the solution with the least amount of steps
    - Solution 2 performs the worst. Worst case is if we have 2 very large prime numbers
    - Solution 3’s performance depends on how we prime factor the numbers

- When asked to design an algorithm…
  - present key observations
  - describe algorithm
  - write code or pseudocode
  - explain why it works and check for correctness
  - analyze time complexity

- Important problem types:
  - sorting (key, stable, in-place, etc.)
  - searching
  - string processing (matching, etc.)
  - graph problems (traversal, shortest path, topological sort, TSP, graph-coloring, etc.)
  - combinatorial problems
  - geometric problems (convex-hull, closest pair, etc.)
  - numerical problems (equation-solving, etc.)