# Programming, Data Structures, and Algorithms in Python: Week 1

This document provides a comprehensive summary of the concepts introduced in the first week of the course, covering algorithms, programming fundamentals, and the practical application of these concepts through the problem of finding the Greatest Common Divisor (GCD).

## 1. Introduction to Algorithms and Programming

### What is an Algorithm?

An algorithm is a systematic, step-by-step description of how to perform a task or solve a problem. Think of it as a recipe: it lists the necessary ingredients (the data) and provides a sequence of instructions to achieve a desired outcome.

The key requirements for an algorithm are:

1. **Finiteness:** The description of the algorithm must be finite.
2. **Termination:** The algorithm must be guaranteed to finish after a finite number of steps for any valid input.

### What is Programming?

Programming is the process of translating an algorithm into a language that a computer can understand and execute.

* **Program:** A program is the implementation of an algorithm.
* **Programming Language:** A formal language used to write programs, like Python.

Our focus is on **computer algorithms**, which primarily manipulate information. This isn't limited to numerical calculations like finding a square root; it also includes tasks like sorting data, finding the best route between two cities, or suggesting spelling corrections.

## 2. The Greatest Common Divisor (GCD)

To illustrate these concepts, we'll use a classic problem: finding the **Greatest Common Divisor (GCD)** of two positive integers, m and n.

* A **divisor** k of m is a number that divides m with no remainder.
* A **common divisor** of m and n is a number that divides both m and n.
* The **GCD** is the largest of these common divisors.

**Examples:**

* gcd(8, 12) is 4.
* gcd(18, 25) is 1 (since 1 is the only common divisor).

### Approach 1: The Naive, Definition-Based Algorithm

This method follows the definition of GCD literally.

**Algorithm:**

1. Generate a list of all factors for m (from 1 to m).
2. Generate a list of all factors for n (from 1 to n).
3. Compare the two lists and find the largest number that appears in both.

Python Implementation:

This translates into a Python program that uses lists to store factors and loops to find them.

def gcd\_naive(m, n):  
 """Computes GCD using the naive list-based approach."""  
 fm = [] # Factors of m  
 for i in range(1, m + 1):  
 if m % i == 0:  
 fm.append(i)  
  
 fn = [] # Factors of n  
 for j in range(1, n + 1):  
 if n % j == 0:  
 fn.append(j)  
  
 cf = [] # Common factors  
 for f in fm:  
 if f in fn:  
 cf.append(f)  
  
 # The last element in the list of common factors is the largest  
 return cf[-1]

### Approach 2: Improving the Naive Algorithm

The first approach is inefficient. We can make several improvements.

1. **Single Scan:** Instead of two separate scans, we can scan from 1 up to the **minimum** of m and n. A number larger than min(m, n) cannot be a common divisor.
2. **No Lists Needed:** We don't need to store all common factors. We only need the *largest* one. We can keep track of the "most recent common factor" found so far.
3. **Scan Backwards:** The most significant optimization to this method is to scan **backwards** from min(m, n) down to 1. The very first common factor we find is guaranteed to be the greatest. This allows us to stop immediately.

Python Implementation (Scanning Backwards):

This version introduces the while loop, which continues as long as a condition is true.

def gcd\_scan\_backwards(m, n):  
 """Computes GCD by scanning backwards."""  
 i = min(m, n)  
 while i > 0:  
 if m % i == 0 and n % i == 0:  
 return i # First common factor found is the GCD  
 i = i - 1

This is simpler and more direct, but its performance is still proportional to the input numbers, which can be very slow for large values.

## 3. Euclid's Algorithm: A Dramatically Better Approach

Discovered by the ancient Greek mathematician Euclid, this is one of the oldest and most efficient algorithms known.

### The Core Idea (Subtraction-Based)

The algorithm is based on a simple but powerful property:

If d divides both m and n (where m > n), then d must also divide their difference, m - n.

This means that gcd(m, n) = gcd(n, m - n). We can repeatedly apply this rule to reduce the numbers until the problem becomes trivial.

### The Modern Version (Remainder-Based)

A more efficient version of this idea uses remainders instead of subtraction. The property is:

gcd(m, n) = gcd(n, r), where r is the remainder when m is divided by n.

This is significantly faster because the numbers decrease much more rapidly.

**Algorithm:**

1. Assume m >= n. If not, swap them.
2. As long as n does not divide m (i.e., the remainder is not 0):
   * Replace m with n.
   * Replace n with the remainder of the old m divided by the old n.
3. Once the remainder is 0, the current value of n is the GCD.

Python Implementation (Euclid's Algorithm):

This is the most efficient version. Its runtime is proportional to the number of digits in the input numbers, not their magnitude.

def gcd\_euclid(m, n):  
 """Computes GCD using Euclid's algorithm (remainder-based)."""  
 if m < n:  
 # Ensure m >= n  
 (m, n) = (n, m)  
  
 while m % n != 0:  
 (m, n) = (n, m % n)  
  
 return n

For a number like 1 billion (109), the naive algorithm might take billions of steps, whereas Euclid's algorithm would take only a few dozen.

## 4. Fundamental Programming Concepts

Our exploration of GCD illustrated several core programming concepts:

* **Variables & Assignment:** Using names (m, n, fm, cf, i) to store and update values. The equals sign (=) is the **assignment operator**.
* **Data Structures:** A way to organize collections of data. We used a **list** ([]) to store factors.
* **Control Flow:** The order in which steps are executed.
  + **Conditional Steps (if):** Execute a block of code only if a certain condition is true (e.g., if m % i == 0).
  + **Repeated Steps (Loops):**
    - for loop: Repeats a block of code for a fixed number of iterations (e.g., for each number from 1 to m).
    - while loop: Repeats a block of code as long as a condition remains true.
* **Recursion:** A function that solves a problem by calling itself with smaller or simpler inputs. The first version of Euclid's algorithm can be naturally expressed this way.

## 5. Getting Started with Python

### Python 2 vs. Python 3

* **Python 2.7:** An older, legacy version.
* Python 3: The modern, actively developed version.  
  This course uses Python 3. While there are many similarities, Python 3 introduced changes that make it more consistent and robust.

### Installation

Python is available for all major platforms (Linux, macOS, Windows).

* **Linux:** Often pre-installed. Use your package manager to install python3 if needed.
* **macOS/Windows:** Download the official installer from [python.org](https://www.python.org/downloads/).

### Interpreter vs. Compiler

* **Compiler:** Translates an entire high-level program into a low-level machine-executable file.
* **Interpreter:** Reads and executes a high-level program line by line.

Python is primarily an **interpreted language**. This allows for an interactive workflow where you can type commands and see immediate results.

### Using the Python Interpreter

1. Open your terminal or command prompt.
2. Type python3 (or python depending on your system) and press Enter.
3. You will see the interpreter prompt: >>>.
4. You can now type Python code directly.

Loading Code from a File:

For larger programs, you type the code into a text file (with a .py extension) and load it into the interpreter.

# In the Python interpreter  
>>> from my\_gcd\_file import \*  
>>> gcd\_euclid(99999, 10000)

### Further Resources

* **Official Python Tutorial:** [docs.python.org/3/tutorial/](https://docs.python.org/3/tutorial/)
* **Dive Into Python 3:** An online book for those with some programming experience.
* **Think Python:** An online book focused on computational thinking using Python.