Lecture Unit 6: Two Fundamental Algorithms
Outline of This Lecture

❖ Euclidean Algorithm for Computing the Greatest Common Divisor

❖ Newton-Raphson Algorithm for Computing Zeros of Functions
Greatest Common Divisor $\text{gcd}(m,n)$

**Definition:** Given positive integers $m$ and $n$, their *greatest common divisor* $\text{gcd}(m,n)$ is the largest positive integer that divides (without remainder) both $m$ and $n$.

**Fundamental Theorem of Arithmetic:** Every positive integer has unique *prime factorization*, it can be always written in one and only one way as a product of primes.

**High-school method for computing the GCD:**

- Compute the *prime factorization* of both $m$ and $n$.
- Then $\text{gcd}(m,n)$ is the product of all the common prime factors.

**Example:** $m = 700$ and $n = 270$

$m = 2^2 \cdot 5^2 \cdot 7$

$n = 2 \cdot 3^3 \cdot 5$

$\text{gcd}(700, 270) = 2 \cdot 5 = 10$
What's Wrong with Prime Factorization?

❖ Computing \( \gcd(m,n) \) via the prime factorization of \( m \) and \( n \) works well if both \( m \) and \( n \) are reasonably small. However, what is the prime factorization of the following number:

\[
1350664108659952233496032162788059699388814756056670 \\
2752448514385152651060485953383940287150571909441 \\
798207282164471551373680419703964191743046496589274 \\
2562393410208643832021103729587257623585096431105640 \\
7350150818751067659462920556368552947521350085287941 \\
6377328533906109750544334999811150056977236890927563
\]

Nobody knows! This number has 309 decimal digits (1024 bits), and the computational hardness of factoring such numbers is the basis for modern cryptosystems that underly e-commerce, national security, etc. The number above is known as RSA-1024, and up until the year 2007 RSA Laboratories Inc. offered a prize of $100,000 for its factorization.

The Euclidean algorithm, which is 2400 years old, efficiently computes \( \gcd(m,n) \) even when \( m \) and \( n \) are both as large as RSA-1024.
Suppose $m > n$. If $n$ divides $m$, then obviously $\gcd(m,n) = n$. Otherwise, let $d$ be any divisor of both $m$ and $n$. Then we have:

\[
\begin{align*}
  m - 2n & \quad m - n & \quad m \\
  n & & d
\end{align*}
\]

**Conclusion:** If $d$ divides both $m$ and $n$, then it also divides $m-n$, and $m-2n$, and $m-3n$, and so on... up to $m$ modulo $n$. Thus:

\[
\gcd(m,n) = \gcd(n,m \% n)
\]
The Euclidean Algorithm

Algorithm: Repeatedly compute "the larger number modulo the smaller number" until this computation yields zero:

\[ m = q_1 \cdot n + r_1 \quad \text{where} \quad r_1 = m \mod n \quad \Rightarrow \quad \text{gcd}(m,n) = \text{gcd}(n,r_1) \]

\[ n = q_2 \cdot r_1 + r_2 \quad \text{where} \quad r_2 = n \mod r_1 \quad \Rightarrow \quad \text{gcd}(n,r_1) = \text{gcd}(r_1,r_2) \]

\[ r_1 = q_3 \cdot r_2 + r_3 \quad \text{where} \quad r_3 = r_1 \mod r_2 \quad \Rightarrow \quad \text{gcd}(r_1,r_2) = \text{gcd}(r_2,r_3) \]

\[ \vdots \]

\[ r_{i-2} = q_i \cdot r_{i-1} + r_i \quad \text{where} \quad r_i = r_{i-2} \mod r_{i-1} \quad \Rightarrow \quad \text{gcd}(r_{i-2},r_{i-1}) = \text{gcd}(r_{i-1},r_i) \]

\[ r_{i-1} = q_{i+1} \cdot r_i + 0 \quad \text{where} \quad r_{i+1} = 0 \quad \Rightarrow \quad \text{gcd}(m,n) = \text{gcd}(r_{i-1},r_i) = r_i \]
Euclidean Algorithm: Examples

Example: \( m = 700 \) and \( n = 270 \)

\[
\begin{array}{c|c}
  m & n \\
  \hline
  700 & 270 \\
  270 & 160 \\
  160 & 110 \\
  110 & 50 \\
  50 & 10 \\
  10 & 0 \\
\end{array}
\]

\[
\text{gcd}(700, 270) = 10
\]

Example: \( m = 100, n = 17 \)

\[
\begin{array}{c|c}
  m & n \\
  \hline
  100 & 17 \\
  17 & 15 \\
  15 & 2 \\
  2 & 1 \\
  1 & 0 \\
\end{array}
\]

\[
\text{gcd}(100, 17) = 1
\]
```c
#include <stdio.h>

int main()
{
    unsigned int m, n, temp;

    printf("Enter two positive integers: ");
    scanf("%u%u", &m, &n);

    while (n != 0)
    {
        temp = n;
        n = m % n;
        m = temp;
    }

    printf("The gcd is %d\n", m);
    return 0;
}
```
Outline of This Lecture

- Euclidean Algorithm for Computing the Greatest Common Divisor
- Newton-Raphson Algorithm for Computing Zeros of Functions
Definition of the Problem: Given a continuous differentiable function \( f(x) \), solve the equation \( f(x) = 0 \). That is, find a point where the function \( f(x) \) intersects the \( x \)-axis.

Newton-Raphson Algorithm: Solves this problem efficiently for an arbitrary function \( f(x) \). Given any initial guess \( x_0 \), the algorithm iteratively computes a sequence of points \( x_0, x_1, x_2, x_3, \ldots \) that quickly approach the point where \( f(x) \) intersects the \( x \)-axis.
Newton-Raphson: Basic Idea

**Taylor Series Expansion:** Every function $f(x)$ that is continuous and differentiable at the point $x_0$ can be expressed as follows:

$$f(x_0 + \Delta) = f(x_0) + f'(x_0) \Delta + \frac{f''(x_0)}{2!} \Delta^2 + \frac{f'''(x_0)}{3!} \Delta^3 + \cdots$$

*This is very small!*

**Linear Approximation:** When $\Delta$ is small, $\Delta^2$ is much smaller than $\Delta$, and $\Delta^3$ is smaller still, and so on. For example, if $\Delta = 0.01$, then $\Delta^2 = 0.0001$, and $\Delta^3 = 0.000001$, and so on. Therefore $f(x_0 + \Delta)$ can be approximated as:

$$f(x_0 + \Delta) \approx f(x_0) + f'(x_0) \Delta$$

**Solving the Equation:** What should $\Delta$ be so that $f(x_0 + \Delta) = 0$ ...?

$$f(x_0 + \Delta) \approx f(x_0) + f'(x_0) \Delta = 0 \quad \Rightarrow \quad \Delta = -\frac{f(x_0)}{f'(x_0)}$$
The rule we have developed based upon approximating the Taylor series of a function $f(x)$, namely:

$$x_1 = x_0 + \Delta = x_0 - \frac{f(x_0)}{f'(x_0)}$$

also has a simple geometric interpretation in terms of approximating the function $f(x)$ by its **tangent at the point** $x_0$, as follows:

$$\tan \theta = \frac{f(x_0)}{x_0 - (x_0 + \Delta)} = \frac{f(x_0)}{-\Delta} = \frac{f(x_0)}{f(x_0)/f'(x_0)} = f'(x_0)$$
The Newton-Raphson Algorithm

Given a continuous differentiable function $f(x)$ and an initial point $x_0$, the Newton-Raphson algorithm generates the sequence of points:

$$x_1 = x_0 + \Delta_0 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

$$x_2 = x_1 + \Delta_1 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

$$x_3 = x_2 + \Delta_2 = x_2 - \frac{f(x_2)}{f'(x_2)}$$

$$\vdots$$

$$x_{i+1} = x_i + \Delta_i = x_i - \frac{f(x_i)}{f'(x_i)}$$

This sequence of points $x_0, x_1, x_2, x_3, \ldots$ very quickly approaches the exact solution, namely a point $x^\ast$ with $f(x^\ast) = 0$.

**Assumption:** The derivative $f'(x_i)$ is never zero!
Newton-Raphson: Demonstration

The diagram illustrates the Newton-Raphson method for finding roots of a function. The function is represented as $f(x)$, and the method iteratively improves the estimate of the root. The iterative process is visualized by the red lines, each tangent to the function curve at the corresponding point. The points $x_0, x_1, x_2, x_3$ represent successive approximations to the root, with each iteration bringing the estimate closer to the actual root of the function.

The $x$-axis represents the variable $x$, and the $y$-axis represents $f(x)$. The goal is to find the value of $x$ where $f(x) = 0$. The Newton-Raphson method uses the derivative of the function to calculate the next approximation, as given by the formula $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.
When Does the Algorithm Stop?

Method ❶: \(|x_{i+1} - x_i| < \varepsilon\)

Method ❷: \(|f(x_{i+1})| < \varepsilon\)

Method ❸: *Fixed number of iterations*
Newton-Raphson: General Outline

Input to the Newton-Raphson Algorithm

- An arbitrary differentiable function \( f(x) \).
- The derivative \( f'(x) \) of this function.
- The initial point \( x_0 \).

The Main Loop of Newton-Raphson

- Recall that the algorithm just iterates
- We can implement it like this:

```c
x = x0;
for (i = 0; i < MAX_ITERATIONS; i++)
{
    fx = f(x);
    fdx = f'(x);
    x = x - fx/fdx; /* what if fdx == 0 ...? */
}
```

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}
\]
Coding Example for \( f(x) = x^2 - 9 \)

```c
#include <stdio.h>
define MAX_ITERATIONS 20

int main()
{
    double x, fx, fdx; int i;
    printf("Enter new initial point: ");
    scanf("%lf", &x);
    for (i = 0; i < MAX_ITERATIONS; i++)
    {
        fx = x*x - 9;
        fdx = 2*x;
        if (fdx == 0) break;
        x = x - fx/fdx;
        printf("Iteration: %d, Solution: %.12f\n", i, x);
    }
    printf("Best solution is: %.12f\n", x);
    return 0;
}
```

Do I need to write a different program for \( \ln(x) \) and another one for \( \sin(x^5) \)?

Functions, and pointers to functions!