Outline

❖ Integers and binary representation
  ▸ Square root of integer
  ▸ Base change

❖ Basic number theory
  ▸ Primality testing and factorization
  ▸ Greatest common divisor and Euclidean Algorithm

❖ Iterative methods
  ▸ Function roots and Newton-Raphson Algorithm
  ▸ Square root
  ▸ Approximating $\pi$
  ❖ Structure & Style

No new C material
Apply what learned to math
Powerful algorithms
Square Root

❖ Find \( \lceil \sqrt{n} \rceil \) (ceiling of square root of \( n \))

- 9 \( \rightarrow \) 3
- 10 \( \rightarrow \) 4
- 16 \( \rightarrow \) 4
- 17 \( \rightarrow \) 5

First int whose square is \( \geq n \)

#include <stdio.h>  
sqrt1.c

int main() {
    int n, i=0;
    printf("Nonnegative integer: ");
    scanf("%d", &n);
    while (i*i<n)
        i++;
    printf("ceiling(sqrt)=%d\n", i);
}

❖ Find \( \lfloor \sqrt{n} \rfloor \) (floor of square root of \( n \))

- 9 \( \rightarrow \) 3
- 10 \( \rightarrow \) 3
- 16 \( \rightarrow \) 4
- 17 \( \rightarrow \) 4

Last int whose square is \( \leq n \)

#include <stdio.h>  
sqrt2.c

int main() {
    int n, i=0;
    printf("Nonneg. integer: ");
    scanf("%d", &n);
    while (i*i<=n)
        i++;
    printf("floor(sqrt)=%d\n", i-1);
}

❖ Better methods when study functions
Digit Reversal

❖ Print an integer in reverse

```c
#include <stdio.h>

int main() {
    int n;
    printf("Positive integer: ");
    scanf("%d", &n);
    do
        printf("%d", n%10);
    while (n/=10);
    printf("\n");
    return 0;
}
```

Print last digit

Some digits left

Remove last digit
The Binary Number System

❖ **Decimal**: digits \{0,1,2,3,4,5,6,7,8,9\} and powers of 10

\[
\begin{array}{cccc}
9 & 3 & 7 & 5 \\
\end{array}
\]

\[
\begin{array}{cccc}
10^3 & 10^2 & 10^1 & 10^0 \\
\end{array}
\]

\[
\begin{array}{c}
9000 + 300 + 70 + 5 = 9375
\end{array}
\]

❖ **Binary**: binary digits (bits) \{0,1\} and powers of 2

\[
\begin{array}{cccc}
1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
2^3 & 2^2 & 2^1 & 2^0 \\
\end{array}
\]

\[
\begin{array}{c}
8 + 0 + 2 + 0 = 10
\end{array}
\]
#include <stdio.h>

int main() {
    int binary, power=1, decimal=0;
    printf("Binary number: ");
    scanf("%d", &binary);
    printf("(%d)_2 = ", binary);
    do {
        decimal += power*(binary%10);
        power*=2;
    } while (binary/=10);
    printf("(%d)_10\n", decimal);
    return 0;
}
Decimal to Binary

❖ What is decimal 10 in binary?

Method 1

<table>
<thead>
<tr>
<th>10</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0 \]
\[ 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1 \]

 Outputs bits from left to right, implement next

❖ Same answer, different method:

Method 2

quotient

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
</table>

\[ \text{div:2} \]

remainder

| 1 |

\[ \text{div:2} \]

remainder

| 2 |

\[ \text{div:2} \]

remainder

| 5 |

\[ \text{div:2} \]

remainder

| 10 |

\[ \text{Outputs bits from right to left, requires memory, after arrays} \]
#include <stdio.h>

int main() {
    int decimal, power=1;

    printf("Positive decimal number: ");
    scanf("%d", &decimal);
    printf("(%d)_10 = (", decimal);

    while (power<=decimal)
        power *= 2;

    while (power/=2) {
        printf("%d", decimal/power);
        decimal%=power;
    }

    printf(")_2\n");
    return 0;
}

Method 2 produces bits right to left, requires memory - after arrays

want to print e.g. (9)\_10 = (1001)\_2

**power**: least power of 2 that is > decimal

halve power

1 if what’s left >= current power, 0 else

keep remainder
Outline

❖ Integers and binary representation
  ▪ Square root of integer
  ▪ Base change

❖ Basic number theory
  ▪ Primality testing and factorization
  ▪ Greatest common divisor and Euclidean Algorithm

❖ Iterative methods
  ▪ Function roots and Newton-Raphson Algorithm
  ▪ Square root
  ▪ Approximating π
**Definition:** An integer \( n \geq 2 \) is **prime** if it is divisible only by itself and 1, namely none of 2, 3, ..., \( n-1 \) divides it. Conversely, \( n \geq 2 \) is **not prime** if some of 2, 3, ..., \( n-1 \) divides it.

**Problem:** Decide whether a given positive integer \( n \) is **prime**

**Program:**

```c
int isPrime=1, n, i;

printf("Test: ");
scanf("%d", &n);

if (n <= 1)
    isPrime = 0;
else
    for (i = 2; i < n; i++)
        if (n % i == 0) isPrime = 0;

if (isPrime)
    printf("%d is a prime\n", n);
else
    printf("%d is not a prime\n", n);

printf("%d is%s a prime\n", n, isPrime ? "" : " not");
```

Primes
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...

Not primes
0, 1, 4, 6, 8, 9, 10, 12, ...

\( \leq 1 \) is not a prime
divisible by \# between 2 and \( n-1 \)
int isPrime = 1, n, i, sqrt_n=0, j;

if (n <= 1 || (n > 2 && n%2 == 0))
    isPrime = 0;
else {
    while ((j=sqrt_n+1)*j<=n)
        sqrt_n++;

    for (i=3; i<=sqrt_n && isPrime; i+=2)
        if (n%i == 0)
            isPrime = 0;
}

Programs should be:
Correct
Efficient
Pretty
A **prime factorization** is an expression of an integer as a product of primes.

- $8 = 2 \times 2 \times 2$
- $12 = 2 \times 2 \times 3$
- $36 = 2 \times 2 \times 3 \times 3$
- $75 = 3 \times 5 \times 5$
- $7 = 7$

**Fundamental Theorem of Arithmetic:** Every integer $\geq 2$ has a unique **prime factorization** - it can always be written in one and only one way as a product of primes.
Factor an integer >= 2 into primes

```c
#include <stdio.h>

int main() {
    int n, i = 2;
    printf("# to factor: ");
    scanf("%d", &n);
    printf("%d =", n);
    while (n > 1)
        if (n % i == 0) {
            printf(" %d *", i);
            n /= i;
        }
    else
        i++;
    printf("\b \n");
    return 0;
}
```

- `i` is a factor
- Remains to be factored
- Erase final *
Complexity of Factorization

❖ Factorizing relatively small numbers - easy
❖ How about:

```
135066410865995223349603216278805969938881475605667027524485
1 4 3 8 5 1 5 2 6 5 1 0 6 0 4 8 5 9 5 3 3 8 3 3 9 4 0 2 8 7 1 5 0 5 7 1 9 0 9 4 4 1
798207282164471551373680419703964191743046496589274
256239341020864383202110372958725762358509643110564073501508
187510676594629205563685529475213500852879416377328533906109
7550544334999841150056977236890927563
```

Nobody knows!

❖ 309 decimal digits (1024 bits) and is known as RSA-1024
❖ Till 2007 RSA Laboratories Inc. offered $100,000 for its factorization
❖ Hardness of factoring large numbers is basis of modern cryptosystems underlying e-commerce, national security, etc.
Greatest Common Divisor

**Definition**
The *greatest common divisor* of two integers \( m \) and \( n \), denoted \( \text{gcd}(m,n) \), is the largest integer dividing both.

\[
\begin{align*}
gcd(9, 6) &= 3 \\
gcd(24, 36) &= 12 \\
gcd(72, 27) &= 9
\end{align*}
\]

**Applications**
- Reduce fractions to smallest terms
- Cryptography, communication, ...

**3 successively faster algorithms**
Brute Force

Start with smaller of \( m, n \), search down till you find a divisor of both

\[ \text{gcd}(10, 6) \]

6

Doesn’t divide 10

5

Doesn’t divide 6

4

Doesn’t divide either

3

Doesn’t divide 10

2

Bingo!

```
#include <stdio.h>

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

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

    for (i=m<n ? m : n; m%i || n%i; i--);
    printf("gcd = %u\n", i);
    return 0;
}
```

Problem: \( \text{gcd}(1 000 000 000, 1 000 000 007) \)

Can take a loooong time
Prime Factorization

❖ Prime factorize \(m\) and \(n\)

❖ \(\text{gcd}(m,n)\) is the product of all \textit{common} factors

\[
gcd(700,540) \quad 700 = 2^2 \times 5^2 \times 7 \quad 540 = 2^2 \times 3^3 \times 5 \quad \text{gcd}(700, 540) = 2^2 \times 5 = 20
\]

❖ Works well if \(m, n\) are small

❖ May not work well for large \(m, n\) (recall RSA 1024):

\[
135066410865995223349603216278805969938881475605667027524485143851526510604859533833940287150571909441798207282164471551373680419703964191743046496589274256239341020864383202110372958725762358509643110564073501508187510676594629205563685529475213500852879416377328533906109750544334999811150056977236890927563
\]

Nobody knows!
The Euclidean algorithm, ~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 clearly $\text{gcd}(m,n) = n$. Otherwise, let $d$ be any divisor of both $m$ and $n$.

Then $d$ also divides

$$m \quad m - n \quad m - 2n$$

$n \quad n$

What remains in the end is $m \% n$

**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 $n$. And conversely, hence:

$$\text{gcd}(m, n) = \text{gcd}(n, m \% n)$$
The Euclidean Algorithm

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

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

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

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

\[ \vdots \]

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

\[ r_{i-1} = q_{i+1} \times r_i + 0 \quad \text{where} \quad r_{i+1} = 0 \quad \Rightarrow \quad \gcd(r_{i-1},r_i) = r_i \]

\[ \gcd(m,n) = \gcd(n,r_1) = \gcd(r_1,r_2) = \ldots = \gcd(r_{i-2},r_{i-1}) = \gcd(r_{i-1},r_i) = r_i \]
### Example: \( m = 700 \) and \( n = 270 \)

<table>
<thead>
<tr>
<th></th>
<th>( m )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>700</td>
<td>270</td>
</tr>
<tr>
<td>2</td>
<td>270</td>
<td>160</td>
</tr>
<tr>
<td>3</td>
<td>160</td>
<td>110</td>
</tr>
<tr>
<td>4</td>
<td>110</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ m = q \times n + r \]

\[ 700 = 2 \times 270 + 160 \]

\[ 270 = 1 \times 160 + 110 \]

\[ 160 = 1 \times 110 + 50 \]

\[ 110 = 2 \times 50 + 10 \]

\[ 50 = 5 \times 10 + 0 \]

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

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

<table>
<thead>
<tr>
<th></th>
<th>( m )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

int main() {
    unsigned int large, small, remainder;
    printf("Two positive integers: ");
    scanf("%u%u", &large, &small);
    do {
        remainder = large % small;
        large = small;
        small = remainder;
    } while (small != 0);
    printf("gcd = %u\n", large);
    return 0;
}
Outline

❖ Integers and binary representation
  ▸ Square root of integer
  ▸ Base change

❖ Basic number theory
  ▸ Primality testing and factorization
  ▸ Greatest common divisor and Euclidean Algorithm

❖ Iterative methods
  ▸ Function roots and Newton-Raphson Algorithm
  ▸ Square root
  ▸ Approximating π
**Problem Definition:** Given a continuous differentiable function $f(x)$, solve the equation $f(x) = 0$. Namely, find a point where the function $f$ intersects the $x$-axis.

**Newton-Raphson Algorithm:** Given an initial guess $x_0$, iteratively computes a sequence of points $x_0, x_1, x_2, x_3, \ldots$ that approach a point where $f$ intersects the $x$-axis.
Given a point $x_0$, can we find another point closer to a root?

Draw line tangent to $f$ at $(x_0, f(x_0))$

$x_1$ - location where tangent intersects $x$ axis

$x_1$ - typically closer to a root than $x_0$

**Formula**

$$f'(x_0) = \tan \theta = \text{slope} = \frac{f(x_0)}{x_0 - x_1}$$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$
The Newton-Raphson method is demonstrated in the following equation:

\[ x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \]
The Newton-Raphson Algorithm

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

- $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$
- $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$
- $x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$
- $\vdots$
- $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$

$x_0, x_1, x_2, x_3, \ldots$ approach a root - a point $x$ with $f(x) = 0$

**Assumption:** The derivative $f'(x_i)$ is not zero
When to Stop?

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

Method 2: \( |f(x_{i+1})| < \varepsilon \)

Method 3: *Fixed number of iterations*
Input to the Newton-Raphson Algorithm

- Arbitrary differentiable function $f(x)$
- Derivative $f'(x)$ of this function
- Initial point $x_0$.

Main Loop of Newton-Raphson

- Iteration: $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
- Can implement as

```c
x = x0;
for (i = 0; i < NUM_ITERATIONS; i++) {
    fx = f(x);
    dfx = f'(x);
    x -= fx/dfx; /* what if dfx == 0 ...? */
}
```
#include <stdio.h>
#define NUM_ITERATIONS 20

int main() {
    double x, fx, dfx; int i;

    printf("Initial point: ");
    scanf("%lf", &x);

    for (i = 0; i < NUM_ITERATIONS; i++) {
        fx = x*x - 9;
        dfx = 2*x;

        if (dfx == 0)
            break;

        x -= fx/dfx;
        printf("x_%d = %.12f\n", i, x);
    }

    printf("Approximately: %.12f\n", x);
    return 0;
}
How to calculate $\sqrt{a}$?

$\sqrt{a} = x$ iff $x^2 = a$ iff $x^2 - a = 0$

$f(x) = x^2 - a$

df = 2x

Other ways too

Algorithms like N-R that successively approximate the solution are called \textit{iterative}. We will next see another.

```c
#include <stdio.h>

int main() {
    double a, x=1, fx, dfx;
    int i;

    printf("Square root of: ");
    scanf("%lf", &a);
    for (i = 0; i < 20; i++) {
        fx = x*x - a;
        dfx = 2*x;
        if (dfx == 0)
            break;
        x -= fx/dfx;
    }

    printf("is \sim %.12f\n", x);
    return 0;
}
```
How many digits of $\pi$ do you know?

Welcome to the first ever PI WORLD RANKING LIST
Approximating π

Pie.
3.

I wish I could determine π
1 4 1 5 9 2

Eureka cried the great inventor
6 5 3 5 8

Christmas pudding Christmas pie
9 7 9 3

is the problem’s very center
2 3 8 4 6
Archimedes-Pfaff TT Approximation

Perimeter of circumscribed n-gon: $a_n$

Perimeter of a circle: $2\pi r = \pi$

Perimeter of circumscribing n-gon: $b_n$

For all $n$: $a_n < \pi < b_n$

As $n$ increases, approximation improves

$\frac{1}{2}$

$\frac{1}{2}$

Johann Friedrich Pfaff

Archimedes

$b_4 = 4$

$a_4 = 2\sqrt{2}$

$b_8 = ?$

$a_8 = ?$

$b_{2n} = \frac{2a_nb_n}{(a_n+b_n)}$

$a_{2n} = \sqrt{a_nb_{2n}}$
```c
#include <stdio.h>
#define SR_ITER 35
#define PI_ITER 30
int main () {
    double a, b=4, x=1, fx, dfx;
    int i, j;
    for (j = 0; j < SR_ITER; j++) {
        fx  = x*x - 2;
        dfx = 2*x;
        if (dfx == 0) break;
        x -= fx/dfx;
    }
    a = 2*x;
    printf("a=%.15f  b=%.15f\n", a, b);
    for (i=0; i<PI_ITER; i++) {
        b = 2*a*b/(a+b);
        x=1;
        for (j=0; j < SR_ITER; j++) {
            fx  = x*x - a*b;
            dfx = 2*x;
            if (dfx == 0) break;
            x -= fx/dfx;
        }
        a = x;
        printf("a=%.15f  b=%.15f\n", a, b);
    }
    return 0;
}
```

### Calculating \( a_4 \)
\[ a_4 = 2\sqrt{2} \]

### Calculating \( b_4 \)
\[ b_4 = 4 \]

### Formula for \( b_{2n} \)
\[ b_{2n} = 2a_nb_n/(a_n+b_n) \]

### Formula for \( a_{2n} \)
\[ a_{2n} = \sqrt{a_nb_{2n}} \]

Much simpler with functions

- \( \sqrt{2} \)
Outline

❖ Integers and binary representation
  ▶ Square root of integer
  ▶ Base change

❖ Basic number theory
  ▶ Primality testing and factorization
  ▶ Greatest common divisor and Euclidean Algorithm

❖ Iterative methods
  ▶ Function roots and Newton-Raphson Algorithm
  ▶ Square root
  ▶ Approximating π

❖ Structure & Style

No new C material
Apply what learned to math
Powerful algorithms
Structure & Style

❖ Structure of C
❖ Style
❖ Bugs
Lexical Elements C

❖ A program is a sequence of **valid characters**

<table>
<thead>
<tr>
<th>Lowercase letters</th>
<th>a b c d e f g h i j ... w x y z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uppercase letters</td>
<td>A B C D E F G H I J ... W X Y Z</td>
</tr>
<tr>
<td>Digits</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Other characters</td>
<td>+ - * / = ( ) { } [ ] &lt; &gt; ` &quot; # ! % &amp; _</td>
</tr>
<tr>
<td>White spaces</td>
<td>blank, tab, newline, etc.</td>
</tr>
</tbody>
</table>

❖ Compiler first executes all compiler directives (lines beginning with # e.g. `#include`, `#define`), then splits sequence of characters into **whitespace** and **tokens**

❖ Six types of tokens: **constants** (5, -4.7), **string constants** ("hello"), **keywords**, **identifiers**, **operators**, and **punctuation**

❖ Describe some next
Keywords

- Reserved words with a **prescribed meaning**. **Cannot** be redefined.

<table>
<thead>
<tr>
<th>auto</th>
<th>break</th>
<th>case</th>
<th>char</th>
<th>const</th>
</tr>
</thead>
<tbody>
<tr>
<td>continue</td>
<td>default</td>
<td>do</td>
<td>double</td>
<td>else</td>
</tr>
<tr>
<td>enum</td>
<td>extern</td>
<td>float</td>
<td>for</td>
<td>goto</td>
</tr>
<tr>
<td>if</td>
<td>inline</td>
<td>int</td>
<td>long</td>
<td>register</td>
</tr>
<tr>
<td>restrict</td>
<td>return</td>
<td>short</td>
<td>signed</td>
<td>sizeof</td>
</tr>
<tr>
<td>static</td>
<td>struct</td>
<td>switch</td>
<td>typedef</td>
<td>union</td>
</tr>
<tr>
<td>unsigned</td>
<td>void</td>
<td>volatile</td>
<td>while</td>
<td></td>
</tr>
<tr>
<td>_Bool</td>
<td>_Complex</td>
<td>_Imaginary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- In addition to keywords, there are others that you can redefine, but should **not** (at least not in this course): `main`, `printf`, `scanf`, ...
Identifiers

❖ Names of a variables, functions, etc.

▷ Letters [A–Z, a–z], digits [0–9], and _

▷ Cannot begin with a digit

▷ Any length, but only first 31 characters matter

❖ Choose meaningful names

hint: 15

\[
s_{15} = ha*0.3 + mg*0.2 + fg*0.5;
\]

Use descriptive names. Don’t worry about length - nobody prints anymore.

\[
\text{score}_\text{ece15} = \text{homework}_\text{average} * 0.30 + \\
\text{midterm}_\text{grade} * 0.20 + \text{final}_\text{grade} * 0.50;
\]

❖ Convention: lower case, words separated by _

▷ \textbf{Java}: Second word starts with upper case \textit{finalGrade}

▷ \textbf{Macros}: All upper case \texttt{CLASS\_SIZE}
Operators and Punctuators

- Perform operations, separate words

<table>
<thead>
<tr>
<th>Category</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+, -, *, /, %, ++, --</td>
</tr>
<tr>
<td>Logical</td>
<td>!, &amp;&amp;,</td>
</tr>
<tr>
<td>Bit</td>
<td>&amp;,</td>
</tr>
<tr>
<td>Assignment</td>
<td>*=, /=, %=, +=, -=, =</td>
</tr>
<tr>
<td></td>
<td>&amp;=,</td>
</tr>
<tr>
<td>Comparison</td>
<td>!=, ==, &lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>Conditional</td>
<td>? :</td>
</tr>
<tr>
<td>Function, Array</td>
<td>()[], &amp;,*</td>
</tr>
<tr>
<td>Pointers</td>
<td>&amp;,*</td>
</tr>
<tr>
<td>Structure</td>
<td>-&gt;, .</td>
</tr>
<tr>
<td>Definition</td>
<td>#, ##</td>
</tr>
<tr>
<td>Punctuator</td>
<td>{ }, , ;</td>
</tr>
</tbody>
</table>
3 Types of Programming Errors (Bugs)

❖ Syntax (compiler) errors
  ‣ Typos, mis-punctuations, grammar errors
  ‣ Program won’t compile
  ‣ Found by compiler, easiest to fix

❖ Run-time errors
  ‣ Division by computed 0, infinite loop
  ‣ Program compiles but crashes or hangs
  ‣ Often also relatively easy

❖ Logical errors
  ‣ Wrong algorithm, initial conditions
  ‣ Program runs, but output is wrong
  ‣ Often most difficult to fix
Four Steps of Software Writing

1. **Analyze**: define input and desired output
2. **Design**: decide on data structures and algorithm
3. **Implement**: code using chosen language
4. **Debug**
   
   The more time spent on Steps 1+2, the easier steps 3+4

For large programs, Step 4 is often the most time consuming
Outline

❖ Integers and binary representation
  ‣ Square root of integer
  ‣ Base change

❖ Basic number theory
  ‣ Primality testing and factorization
  ‣ Greatest common divisor and Euclidean Algorithm

❖ Iterative methods
  ‣ Function Roots and Newton-Raphson Algorithm
  ‣ Square root

❖ Approximating π

nicos new C material
Apply what learned to math
Powerful algorithms