MAT 685: C++ for Mathematicians

Greatest Common Divisors

John Perry

University of Southern Mississippi

Spring 2017

Outline

Outline

gcd?

$\gcd(a, b)$: the **greatest common divisor** of two integers

- largest integer factor d of both a, b
- if $d = 1$, we say a, b **relatively prime**

Examples

- $\gcd(24, -16) = 8$
- $\gcd(24, -15) = 1$

Classic problem

- Choose n
- Choose $a, b \in \{1, \ldots, n\}$
- Let p_n be probability that $\gcd(a, b) = 1$
- Does $\lim_{n \to \infty} p_n$ exist?
  - If so, what is its value?

Classic problem

- Choose n
- Choose $a, b \in \{1, \ldots, n\}$
- Let p_n be probability that $\gcd(a, b) = 1$
- Does $\lim_{n \to \infty} p_n$ exist?
  - If so, what is its value?

Example

Let $n = 8$.

- Possible outcomes: $\binom{8}{2} = \frac{8!}{2!6!} = 28$
- Relatively prime pairs: $(1,2)$, $(1,3)$, ..., $(1,8)$, $(2,3)$, $(2,5)$, $(2,7)$, $(3,4)$, $(3,5)$, $(3,7)$, $(3,8)$, $(4,5)$, $(4,7)$, $(6,7)$, $(7,8)$
- So $p_8 = {}^{18}/_{28} = 9/14 \approx 64.3\%$

Limitations/utility of computer

limitations

- $\#n$ infinite
- computer capacity finite

$\therefore$ cannot find answer

utility

- faster computation
- more accurate computation

$\therefore$ can help w/conjecture

Example

n	10	11	12	13	14	15	16	17	18	19
p_n	.63	.69	.63	.68	.65	.64	.62	.66	.63	.66

Example

n	10	11	12	13	14	15	16	17	18	19
p_n	.63	.69	.63	.68	.65	.64	.62	.66	.63	.66

n	100	200	400	800	1600	3200
p_n	.6087	.6116	.6085	.6086	.6080	.6081

Outline

Analyzing the problem

Tasks:

1. Need a way to select all pairs
2. Need a way to compute gcd for each pair

Analyzing the problem

Tasks:

1. Need a way to select all pairs
2. Need a way to compute gcd for each pair

A *good* solution separates these two tasks!

Technical jargon

- In C++, tasks organized as "functions" or "methods"
- C++ "function" *not quite the same* as mathematical "function"

math $f(8) = 7$ now? $f(8) = 7$ five min hence

C++ $f(8) = 7$ now? m/b $f(8) = 15$ five min hence

Technical jargon

- In C++, tasks organized as "functions" or "methods"
- C++ "function" *not quite the same* as mathematical "function"

  math $f(8) = 7$ now? $f(8) = 7$ five min hence

  C++ $f(8) = 7$ now? m/b $f(8) = 15$ five min hence

- book refers to C++ functions as **procedures**
- data pass to procedure is **argument** or **parameter**

Outline

Procedure template

```
return_type proc_name(type1 arg1, ...) {
  statement1;
  statement2;
  ...
}
```

Procedure template

```
return_type proc_name(type1 arg1, ...) {
  statement1;
  statement2;
  ...
}
```

`return_type` type of data to return; `void` if none

`proc_name` valid identifier name (letter followed by letters/numbers/_)

`typei` type of `argi`

`argi` valid identifier name

`statementj` valid C++ statement

Braces

Every language needs some way to identify a block of instructions

C-style {...}
C, C++, Java

Pascal-style `BEGIN...END`
Pascal, Modula-2, Oberon, Eiffel

Python-style indentation
Python, ...?

Braces define scope

scope? block where identifiers have a particular meaning

local scope identifier's meaning within block

global scope identifier's meaning outside all braces

Braces define scope

scope? block where identifiers have a particular meaning

local scope identifier's meaning within block

global scope identifier's meaning outside all braces

```
long time = 0;
{
  long time = 2;
  cout << time << endl;
}
cout << time << endl;
```

What values are printed out?

Outline

Interface, implementation

interface how function is used/called/invoked

implementation how function behaves

Interface, implementation

interface how function is used/called/invoked

implementation how function behaves

Good programming practice

Separate interface from implementation.

Example

```
/** returns determinant of 2x2 matrix */
long determinant_2x2(
    long a, long b, long c, long d
) {
  return a*d - b*c;
}
```

Example

```
/** returns determinant of 2x2 matrix */
long determinant_2x2(
    long a, long b, long c, long d
) {
  return a*d - b*c;
}
```

In this example,

- **interface** is `int determinant(int, int, int, int)`
- **implementation** is `return a*d - b*c;`

Interface: "header" file

C++ convention: interface in header file, implementation in separate file(s)

- convention: `.hpp` header, `.cpp` implementation(s)
- separation usually *necessary* (linking, double definitions, ...)
- argument names not necessary

Interface: "header" file

C++ convention: interface in header file, implementation in separate file(s)

- convention: `.hpp` header, `.cpp` implementation(s)
- separation usually *necessary* (linking, double definitions, ...)
- argument names not necessary

Listing 2: `gcd.hpp`

```
#ifndef __GCD_HPP_
#define __GCD_HPP_

/**
  Calculate the greatest common divisor
  of two integers.
*/
long gcd(long, long);

#endif
```

"Preprocessor directives"

start with #

`#define` defines a preprocessor symbol
used here to ensure code not imported twice

`#ifndef` "if not defined"
if following symbol *is* defined, everything ignored until...

`#endif` conclude `#ifndef` statement

Question

Where else have you seen preprocessor directives?

Good programming practice

Always, *always* define a preprocessor symbol for header files.

Things to notice: body

- comments document function
- no implementation in header
- interface *need not* specify variables names

Things to notice: body

- comments document function
- no implementation in header
- interface *need not* specify variables names
- *may* specify variable names for documentation

Listing 4: book's `gcd.hpp`

```
/**
  Calculate the greatest common divisor
  of two integers.
  @param a first integer
  @param b second integer
  @return greatest common divisor of a, b
*/
long gcd(long a, long b);
```

Outline

Outline

Euclidean algorithm

given $a, b \in \mathbb{Z}$
if $a = 0$ **return** b
if $b = 0$ **return** a
let r be remainder of a divided by b
return $\gcd(b, r)$

Euclidean algorithm

given $a, b \in \mathbb{Z}$
if $a = 0$ **return** b
if $b = 0$ **return** a
let r be remainder of a divided by b
return $\gcd(b, r)$

$$\begin{aligned}\gcd(26, 14) &= \gcd(14, 12) \\ &= \gcd(12, 2) \\ &= \gcd(2, 0) \\ &= 2.\end{aligned}$$

Why does it work?

Theorem (Euclid; Proposition 3.1, p. 37)

Let $a, b \in \mathbb{N}^+$ *and* $r = a \bmod b$. *Then* $\gcd(a, b) = \gcd(b, r)$.

Why does it work?

Theorem (Euclid; Proposition 3.1, p. 37)

Let $a, b \in \mathbb{N}^+$ *and* $r = a \bmod b$. *Then* $\gcd(a, b) = \gcd(b, r)$.

Proof (see also MAT 521, 523).

By Division Theorem, $a = bq + r$.

Why does it work?

Theorem (Euclid; Proposition 3.1, p. 37)

Let $a, b \in \mathbb{N}^+$ *and* $r = a \bmod b$. *Then* $\gcd(a, b) = \gcd(b, r)$.

Proof (see also MAT 521, 523).

By Division Theorem, $a = bq + r$.

$b = dx, r = dy$?

$$a = bq + r = (dx)q + dy = d(xq + y)$$

Why does it work?

Theorem (Euclid; Proposition 3.1, p. 37)

Let $a, b \in \mathbb{N}^+$ *and* $r = a \bmod b$. *Then* $\gcd(a, b) = \gcd(b, r)$.

Proof (see also MAT 521, 523).

By Division Theorem, $a = bq + r$.

$b = dx, r = dy$?

$$a = bq + r = (dx)q + dy = d(xq + y)$$

$a = dz, b = dy$?

$$r = a - bq = dz - dxq = d(z - xq)$$

Why does it work?

Theorem (Euclid; Proposition 3.1, p. 37)

Let $a, b \in \mathbb{N}^+$ *and* $r = a \bmod b$. *Then* $\gcd(a, b) = \gcd(b, r)$.

Proof (see also MAT 521, 523).

By Division Theorem, $a = bq + r$.

$b = dx, r = dy$?

$$a = bq + r = (dx)q + dy = d(xq + y)$$

$a = dz, b = dy$?

$$r = a - bq = dz - dxq = d(z - xq)$$

$$\text{common divisors of } b, r \iff \text{common divisors of } a, b$$

$$\therefore \gcd(b, r) = \gcd(a, b)$$

C++ Implementation

Listing 5: `gcd_recursive.cpp`

```
#include "gcd.hpp"
#include <iostream>
using std::cerr; using std::endl;

long gcd(long a, long b) {
  // Make sure a, b nonnegative
  if (a < 0) a = -a;
  if (b < 0) b = -b;

  // if a, b both zero print error, return 0
  if ( (a == 0) and (b == 0) ) {
    cerr << "WARNING: gcd called with two zeros.\n";
    return 0;
  }

  if (b == 0) return a;
  if (a == 0) return b;

  long c = a % b;

  return gcd(b, c);
}
```

Example

Compute $\gcd(26, -14)$:

`gcd(26,-14)` `a=26, b=-14`

- $26 \geq 0$ so no change to `a`
- $-14 < 0$ so `b` reassigned to 14
- `a` and `b` not 0
- `c = 26 % 14 = 12`
- return `gcd(14,12)`

Example

Compute $\gcd(26, -14)$:

`gcd(26,-14)`

`gcd(14,12)` `a=14, b=12`

- $14 \geq 0$ so no change to `a`
- $12 \geq 0$ so no change to `b`
- `a` and `b` not 0
- `c = 14 % 12 = 2`
- return `gcd(12,2)`

Example

Compute gcd $(26, -14)$:

`gcd(26,-14)`

`gcd(14,12)`

`gcd(12,2)` `a=12, b=2`

- $12 \geq 0$ so no change to `a`
- $2 \geq 0$ so no change to `b`
- `a` and `b` not 0
- `c = 12 % 2 = 0`
- return `gcd(2,0)`

Example

Compute $\gcd(26, -14)$:

```
gcd(26,-14)
gcd(14,12)
gcd(12,2)
 gcd(2,0)  a=2,  b=0
```

- $2 \geq 0$ so no change to `a`
- $0 \geq 0$ so no change to `b`
- **but** $b = 0$

$\therefore$ `return` $a = 2$

...cascades back up: $\gcd(26, -14) = 2$

A basic test program

Listing 6: `test_gcd_recursive.cpp` — place in same folder as `gcd.*`

```
#include <iostream>
using std::cin; using std::cout;
using std::endl;
#include "gcd.hpp"

int main() {
  long a, b;
  cout << "Enter two numbers: " << endl;
  cin >> a >> b;
  cout << "gcd(" << a << "," << b << ") = ";
  cout << gcd(a,b) << endl;
}
```

Building from command line, running

Method 1 (book's way)

```
$ g++ -o test_gcd gcd_recursive.cpp \
    test_gcd_recursive.cpp
```

- list all *implementation* files
- all files compiled into executable
- `-o` indicates name of executable, `test_gcd`

Building from command line, running

Method 2 (faster in long run)

```
$ g++ -c gcd_recursive.cpp
$ g++ -o test_gcd gcd_recursive.o \
    test_gcd_recursive.cpp
```

- `-c` indicates "compile to object file"
- new file, `gcd_recursive.o`
- second compilation: list *main* file and relevant object files
  - object files not recompiled!
  - *much* faster when changing one file of many

Building from command line, running

Method 2 (faster in long run)

```
$ g++ -c gcd_recursive.cpp
$ g++ -o test_gcd gcd_recursive.o \
    test_gcd_recursive.cpp
```

Usage:

```
$ ./test_gcd
Enter two numbers:
26 14
gcd(26, 14) = 2
```

Outline

Recursion?

Solution to a problem defined in terms of a "simpler" case

Example (Factorials)

$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$, or,

- $1! = 1$
- $n! = n \times (n-1)!$

Recursion?

Solution to a problem defined in terms of a "simpler" case

Example (Factorials)

$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$, or,

- $1! = 1$
- $n! = n \times (n-1)!$

Listing 8: Straightforward implementation of factorials

```
long factorial(long n) {
  if (n == 1) return 0;
  return n * factorial(n - 1);
}
```

Where is the recursion?

Where was the recursion?

```
return n * factorial(n - 1);
```

`factorial()` calls itself!

Good programming practice

Only use recursion if subcases are in some sense "smaller"

Where was the recursion?

```
return n * factorial(n - 1);
```

`factorial()` calls itself!

Good programming practice

Only use recursion if subcases are in some sense "smaller"

Example (Good)

`factorial(10), factorial(9), factorial(8), ...`

Example (Bad. *Very* bad.)

`my_fun(-2), my_fun(-3), my_fun(-4), ...`

Careful with your recursion

Listing 9: Book's first factorial, see p. 39

```
long long factorial(long long n) {
  return n * factorial(n - 1)
}
```

What's wrong with this code?

Careful with your recursion

Listing 12: Book's second factorial, see p. 39

```
long long factorial(long long n) {
  if (n == 0) return 1;
  return n * factorial(n - 1)
}
```

Better, but still buggy. See homework.

Outline

`while` statements

`while` repeats a block of statements as long as some conditions hold

- excellent w/testable condition when task finishes

`while` statements

while repeats a block of statements as long as some conditions hold

- excellent w/testable condition when task finishes

Question

is there a testable condition when task finishes?

`while` statements

while repeats a block of statements as long as some conditions hold

- excellent w/testable condition when task finishes

Question

is there a testable condition when task finishes?

yes: when remainder is 0!

In our case

Listing 13: `gcd_while.cpp`, "..." indicates no change!

```
#include "gcd.hpp"
#include <iostream>
using std::cerr; using std::endl;

long gcd(long a, long b) {
  // Make sure a, b nonnegative
  ...
  // if a, b both zero print error, return 0
  ...

  long new_a, new_b;

  while (b != 0) {
    new_a = b;
    new_b = a % b;
    a = new_a;
    b = new_b;
  }

  return a;
}
```

Example

Compute $\gcd(26, -14)$:

`gcd(26,-14)` `a=26, b=-14`

- $26 \geq 0$ so no change to `a`
- $-14 < 0$ so `b` reassigned to 14
- `a` and `b` not 0
- $b \neq 0$: enter while loop

Example

Compute $\gcd(26, -14)$:

`gcd(26,-14)` `a=26, b=-14`

- $26 \geq 0$ so no change to `a`
- $-14 < 0$ so `b` reassigned to 14
- `a` and `b` not 0
- $b \neq 0$: enter while loop
  - `new_a = 14, new_b = 12`
  - `a = 14, b = 12`
  - $b \neq 0$: continue

Example

Compute $\gcd(26, -14)$:

`gcd(26,-14)` `a=26, b=-14`

- $26 \geq 0$ so no change to `a`
- $-14 < 0$ so `b` reassigned to 14
- `a` and `b` not 0
- $b \neq 0$: enter while loop
  - `new_a = 14,new_b = 12`
  - `a = 14,b = 12`
  - $b \neq 0$: continue
  - `new_a = 12,new_b = 2`
  - `a = 12,b = 2`
  - $b \neq 0$: continue

Example

Compute $\gcd(26, -14)$:

`gcd(26,-14)` `a=26, b=-14`

- $26 \geq 0$ so no change to `a`
- $-14 < 0$ so `b` reassigned to 14
- `a` and `b` not 0
- $b \neq 0$: enter while loop
  - `new_a = 14, new_b = 12`
  - `a = 14, b = 12`
  - $b \neq 0$: continue
  - `new_a = 12, new_b = 2`
  - `a = 12, b = 2`
  - $b \neq 0$: continue
  - `new_a = 2, new_b = 0`
  - `a = 2, b = 0`
  - $b = 0$: exit loop

Example

Compute $\gcd(26, -14)$:

`gcd(26,-14)` `a=26, b=-14`

- $26 \geq 0$ so no change to `a`
- $-14 < 0$ so `b` reassigned to 14
- `a` and `b` not 0
- $b \neq 0$: enter while loop
  - `new_a = 14, new_b = 12`
  - `a = 14, b = 12`
  - $b \neq 0$: continue
  - `new_a = 12, new_b = 2`
  - `a = 12, b = 2`
  - $b \neq 0$: continue
  - `new_a = 2, new_b = 0`
  - `a = 2, b = 0`
  - $b = 0$: exit loop
- return 2

Building from command line, running

Method 1 (book's way)

```
$ g++ -o test_gcd gcd_while.cpp test_gcd_while.cpp
```

- list all *implementation* files
- all files compiled into executable
- `-o` indicates name of executable, `test_gcd`

Building from command line, running

Method 2 (faster in long run)

```
$ g++ -c gcd_while.cpp
$ g++ -o test_gcd gcd_while.o test_gcd_while.cpp
```

- `-c` indicates "compile to object file"
- new file, `gcd_while.o`
- second compilation: list *main* file and relevant object files
  - object files not recompiled!
  - *much* faster when changing one file of many

Building from command line, running

Method 2 (faster in long run)

```
$ g++ -c gcd_while.cpp
$ g++ -o test_gcd gcd_while.o test_gcd_while.cpp
```

Usage:

```
$ ./test_gcd
Enter two numbers:
26 14
gcd(26, 14) = 2
```

Points to ponder

- one header fine, two implementation files!
  - `gcd_recursive.cpp` (recursive)
  - `gcd_while.cpp` (`while`-based)
- can select according to need/desire
- more than one way to solve a problem!

Homework

pp. 49-50 #3.2, 3.3, 3.7

Outline

Outline

Recall the problem

- Choose n
- Choose $a, b \in \{1, \ldots, n\}$
- Let p_n be probability that $\gcd(a, b) = 1$
- Does $\lim_{n \to \infty} p_n$ exist?
  - If so, what is its value?

Recall the problem

- Choose n
- Choose $a, b \in \{1, \ldots, n\}$
- Let p_n be probability that $\gcd(a, b) = 1$
- Does $\lim_{n \to \infty} p_n$ exist?
  - If so, what is its value?

We can now compute $\gcd(a, b)$. How would you choose all pairs $a, b \in \{1, \ldots, n\}$?

What needs to be done?

Need to repeat, so...

- *could* use a `while` loop, but...
- easier way here

What needs to be done?

Need to repeat, so...

- *could* use a `while` loop, but...
- easier way here

  for repeats a block of statements a sort-of-definite number of times

  - excellent when program knows exact quantity of repetitions
  - tool of choice when working with a, $a + 1$, ..., $a + k$
  - also useful over lists, sets, ... (covered later)

Structure

Listing 14: `for` statement structure

```
for (
      initialization ;
      termination condition ;
      advancing statement
) {
   statement1 ;
   statement2 ;
   ...
}
```

(`for` statement can be all on one line)

Remark

`for` statement equivalent to:

```
initialization ;
while (not (termination condition)) {
  statement1 ;
  statement2 ;
  ...
  advancing statement ;
}
```

...but easier to maintain

Classical example

Print numbers from 1 to 10

Listing 15: `print10.cpp`

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  int i;

  for (i = 1; i < 11; ++i) {
    cout << i << endl;
  }
}
```

Result

```
$ g++ -o print10 print10.cpp
$ ./print10
1
2
3
4
5
6
7
8
9
10
```

What about pairs of numbers?

Use a *nested* loop to print $(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)$

Listing 16: `print_pairs.cpp`

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  int i, j;

  for (i = 1; i < 5; ++i) {
    for (j = i + 1; j < 5; ++j) {
      cout << '(' << i << ',' << j << ") ";
    }
  }
  cout << endl;
}
```

What about pairs of numbers?

Use a *nested* loop to print $(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)$

Listing 17: `print_pairs.cpp`

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  int i, j;

  for (i = 1; i < 5; ++i) {
    for (j = i + 1; j < 5; ++j) {
      cout << '(' << i << ',' << j << ") ";
    }
  }
  cout << endl;
}
```

```
$ g++ -o print_pairs print_pairs.cpp
$ ./print_pairs
(1,2) (1,3) (1,4) (2,3) (2,4) (3,4)
```

How does this work?

```
for i = 1
        for j = 2
            prints (1, 2)
            j = 3
            prints (1, 3)
            j = 4
            prints (1, 4)
```

How does this work?

for `i = 1`

    for `j = 2`

        prints $(1, 2)$

        `j = 3`

        prints $(1, 3)$

        `j = 4`

        prints $(1, 4)$

`i = 2`

    for `j = 3`

        prints $(2, 3)$

        `j = 4`

        prints $(2, 4)$

How does this work?

for i = 1
    for j = 2
        prints $(1, 2)$
        j = 3
        prints $(1, 3)$
        j = 4
        prints $(1, 4)$
i = 2
    for j = 3
        prints $(2, 3)$
        j = 4
        prints $(2, 4)$
i = 3
    for j = 4
        prints $(2, 4)$

Alternate way of using `for`

Declare variables when they are used

Listing 18: `print_pairs_inline_decl.cpp`

```
#include <iostream>
using std::cout; using std::endl;

int main() {
  for (int i = 1; i < 5; ++i) {
    for (int j = i + 1; j < 5; ++j) {
      cout << '(' << i << ',' << j << ") ";
    }
  }
  cout << endl;
}
```

Outline

Main program

Listing 19: `prob_relprime_pair.cpp`

```
#include <iostream>
using std::cin; using std::cout; using std::endl;

#include "gcd.hpp"

/** Find probability that two integers in {1,...,n}
    are relatively prime */
int main() {
  long long n;
  cout << "Enter n --> ";
  cin >> n;

  long long count = 0;
  for (long a = 1; a <= n; ++a) {
    for (long b = a + 1; b <= n; ++b) {
      if ( gcd(a,b) == 1 ) ++count;
    }
  }
  count = 2*count + 1;
  cout << double(count) / double(n*n) << endl;
  return 0;
}
```

Compiling, using

Save into directory with `gcd.*`

```
$ g++ -o prob_relprime_pair gcd_while.o \
  prob_relprime_pair.cpp
$ ./prob_relprime_pair
Enter n --> 100
0.6087
$ ./prob_relprime_pair
Enter n --> 800
0.6086
```

Homework

pp. 49–50 #3.4, 3.5, 3.6, 3.8

Outline

Situation

Hypothetical situation

- problem in `gcd_while.cpp`
- eyeball code $\Longrightarrow$ no problem...
- can we step through code?

Solution

`gdb` (interactive debugger)

Invoking debugger

"g++ -g"

Debugger won't help if you leave off "-g"

```
$ g++ -g -c gcd_while.cpp
$ g++ -g -o prob_relprime_pair \
    gcd_while.o prob_relprime_pair.cpp
$ gdb prob_relprime_pair
```

Setting a breakpoint

Problem maybe at or after:

```
while (b != 0) {
```

Setting a breakpoint

Problem maybe at or after:

```
while (b != 0) {
```

- Set breakpoint here (line 19)

```
(gdb) b gcd_while.cpp:19
```

- Start program

```
(gdb) run
```

- Program runs, stops, displays:

```
Breakpoint 1, gcd (a=1, b=2) at gcd_while.cpp:19
19                  while (b != 0) {
(gdb)
```

Yeah, so?

`p e` prints the value of expression *e*

```
(gdb) p b
$1 = 2
(gdb) p b == 0
$2 = false
(gdb) p $1
$3 = 2
```

Yeah, so?

`p e` prints the value of expression *e*

```
(gdb) p b
$1 = 2
(gdb) p b == 0
$2 = false
(gdb) p $1
$3 = 2
```

THIS IS AWESOME

Allows you to investigate, fix *many* problems

Outline

Summary

- Math stuff
  - relatively prime numbers
  - Euclidean Algorithm
- Programming stuff
  - `if` statement (used, not really covered yet)
  - recursion
  - `while` statement
  - for statement
  - debugging variables and expressions