# Notes 7

you've been moved to 24-115, F1 and F2 and you'll be in 36-156 F1.

## 6.5 Finding GCD coefficients

Recall Euclid's algorithm:

**Example 1** Table method for getting GCD and Coefficients: run Euclid's algorithm, then fill in the coefficients from the bottom up.

| q | x | y |
|---|---|---|
| | 949 | 273 |
| 3 | 273 | 130 |
| 2 | 130 | 13 |
| 10 | 13 | 0 |

The algorithm finds the GCD.
It can be modified to give the coefficients in the sum expression.

**Procedure** Euclid-Coefficients($x, y$), where $x$ and $y$ are not both 0, and $x \geq y$

1. If $y = 0$, return $(1, 0)$.
2. Else $x = qy + r$ $(r = x mod y)$
3. Use Euclid-Coefficients($y, r$) to find $(a, b)$ such that GCD$(y, r) = ay + br$
4. Return $(b, a - qb)$.

**Theorem 6.1** If Euclid-Coefficients($x, y$) terminates, its output $(a, b)$ satisfies $ax + by =$ GCD$(x, y)$.

*Proof.* Induction on the number of recursive calls the algorithm takes to terminate.

BASE: no recursion:
Then $y = 0$, the GCD is $x$, and the return values are $(1, 0)$. Since $(1)(x) + (0)(y) = x$, this is correct.

INDUCTIVE STEP:
Show for $n$ passes, assuming OK for $n-1$ passes:

Say $x = qy + r$.
By induction, recursive call returns $(a, b)$ with $ay + br = g$, where $g$ is $\text{GCD}(y, x \bmod y)$.
Remember, this is also $\text{GCD}(x, y)$.
Just rewrite $g$ in terms of $x$ and $y$ instead of in terms of $y$ and $x \bmod y$:
$g = ay + br = ay + b(x - qy) = bx + (a - bq)y$.
Says the return values $(b, a - bq)$ are as claimed. ∎

**Example 2** Table method for getting GCD and Coefficients: run Euclid's algorithm, then fill in the coefficients from the bottom up.

| q | x | y | a | b |
|---|---|---|---|---|
| | 949 | 273 | -2 | 7 |
| 3 | 273 | 130 | 1 | -2 |
| 2 | 130 | 13 | 0 | 1 |
| 10 | 13 | 0 | 1 | 0 |

Filling in table entries from the bottom: $g = (1)(13) + (0)(0)$.
Then $g = (0)(130) + (1)(13)$.
Then $g = (1)(273) + (-2)(130)$.
Then $g = (-2)(949) + (7)(273)$.

## 6.6 RSA cryptosystem

Here's a nice application of number theory to cryptography.
RSA is a *public key* cryptosystem: anyone can encrypt (encode) a message to me, but no one else can decrypt (decode) it.

Uses two *keys*, which are positive integers.
$e$, the *encryption key.* Public.
$d$, the *decryption key.* Private (to the decrypter).

Encryption can be done using just $e$.

Decryption is hard unless you have $d$.

RSA uses modular arithmetic to build $e$ and $d$.

Recall def of mod. There's a different defn:

Lots of properties, see book. one we'll use a lot: $ab \bmod n = (a \bmod n)(b \bmod n) \bmod n)$ (mod floats around).

Another theorem we need: Fermat: if $p$ is prime, then $a^{p-1} = 1 \pmod{p}$.

To do RSA:

- user picks two large primes $p$ and $q$.
- let $n = pq$
- pick (any) encryption key $e$
- pick a related $d$ so that $ed \bmod (p-1)(q-1) = 1$
- publishes $e$ and $n$, keeps $d$.

To encrypt messaage $m$, treat as number.

Build $m^e \bmod n$ (Anyone can do this, because $e$ and $n$ are public. )

When user receieves $m^e$, computes $(m^e)^d = m^{ed} = m^{1+k(p-1)(k-1)} \pmod{n}$.

Fermat says $m^{p-1} = 1 \pmod{p}$, and $m^{q-1} = 1 \pmod{q}$.

Chinese remainder theorem says can multiply: $m^{(p-1)(q-1)} = 1 \pmod{n}$

**Theorem 6.2** Suppose that $p$ and $q$ are primes, $n = pq$, and $e$, $d$ are positive integers with $ed \bmod (p-1)(q-1) = 1$.
Suppose that $m$ is any natural number $< n$ such that (technicality, usually holds) neither $p$ nor $q$ divides $m$.
Then $m^{ed} \bmod n = m$.

*Proof.* In book, see p. 147.
Basically, a bunch of exponential calculations and modular arithmetic. ■

So $m^{ed} = m \pmod{n}$. gets message back.

Example (from book): Let $p = 43$, $q = 59$, so $n = 43 \times 59 = 2537$.
Let $e = 13$, $d = 937$. Then $ed \bmod (p-1)(q-1) = 1$, as needed (check).
Encrypt a number 1819 using exponentiation with 13, mod 2537, get 2081.
Decrypt 2081 using exponentiation with 937, mod 2537, get back 1819. Left to the reader to check this.

Fast exponentiation:
In recitation, we will show how to compute the big exponentiations needed to implement RSA quickly. It relies on the fact that $x^{ab} \bmod n = (x^a \bmod n)^b \bmod n$.

This all works to encrypt and decrypt.
But we haven't addressed the important issue: *secrecy*,
i.e., the property that the coded message is hard to decrypt without having access to $d$, the decryption key. This turns out to depend on whether or not $p$ and $q$ are very large numbers (currently 200 digits), and keeping them, as well as $d$, secret. (Finding $d$ actually lets you find $p$, $q$, and vice versa.)
In case $p$ and $q$ are large, no one yet knows how to "break" RSA:

- No one knows how to compute $p$, $q$ efficiently from $n$. (Factoring large numbers is hard, though there's been progress.)
- No one knows how to compute $d$ efficiently without $p$, $q$.
- No one knows how to find $m$ efficiently without $d$ (ie, solve $c = n^e$ for $m$).

In practice, how do we find suitable $p$, $q$, $e$ and $d$?

There are fast algorithms for generating large prime numbers $p$, $q$. (pick a random number)

For $e$ and $d$:
Recall our objective:
Need $de \bmod k = 1$, where $k = (p-1)(q-1)$.
Pick $e$ relatively prime to $k$ (lots of choices here).
That is, $\mathrm{GCD}(e,k) = 1$.
By GCD theorems, we can find coefficients $a$, $b$ such that $ae + bk = 1$.
Just use $d = a$ – then $de \bmod k = 1$ since $de + bk = 1 \longrightarrow de = -bk + 1 \longrightarrow de \bmod k = 1$.

Used by Netscape, e.g.

## 6.7 Algorithm Verification

Induction can also be used to verify iterative algorithms.

Algorithm for multiplying $a \times b$, where $a$ and $b$ are arbitrary length positive integers represented in binary, and result is to be represented in binary.
Assume that we have integer addition available,
and can examine individual bits.
$b(k)$ denotes $k$th high order bit of $b$.

Algorithm:
Maintain accumulator $acc$, initially set to $a$.
Suppose that $b$ has $n$ bits.

For $k = 1$ to $n - 1$ do:
if $b(k+1) = 0$ then $acc := 2acc$
else $acc := 2acc + a$

At end, result is in $acc$.

Example: $a = 5$, $b = 13 = 1101$.

| $k$ | $acc$ | $high(b, k+1)$ |
|---|---|---|
| 0 | 5 | 1 |
| 1 | 15 | $3 = 11$ |
| 2 | 30 | $6 = 110$ |
| 3 | 65 | $13 = 1101$ |

How do we know this is correct?
Can prove, by induction.
For every $k$, $0 \leq k \leq n-1$, $P(k)$, where
$P(k)$: After $k$ iterations, $acc$ is equal to $a \times high(b, k+1)$.

Writing $high(b, m)$ for the integer defined by the $m$ highest-order bits of $b$.)

BASE: $k = 0$.
The highest-order bit is 1, so $high(b, 1) = 1$.
Initially, $acc$ contains $a = a \times 1$, as needed.
INDUCTIVE STEP:
Assume for $k$. After $k$ iterations, $acc$ contains $a \times high(b, k+1)$.
Show after one more iteration, $acc$ contains $a \times high(b, k+2)$.

Depends on examining the $k+1$st iterative step.
At this step, examine $k+2$nd bit of $b$.
Rule says if 0, double, else double and add $a$.
Check cases to see this is the right thing to do:
If 0, new desired number is just double the old.
If 1, new desired number is double plus $a$, ...

Why does this say that the final answer is correct?

Specialize the result to $k = n-1$, $P(n-1)$:
After $n-1$ iterations, $acc$ is equal to $a \times high(b, n) = b$.

# 7 State Machines

*Everything following is a summary of a more complete version of Handout 11.*
Now start the final discrete structure we will study in this course:
"state machines".

Used as the basis for modelling algorithms and physical machines, in order to give precise specifications of their behavior and in order to verify their properties.

Will present:
Definitions of state machines, executions, reachability.

Basic examples:
Illustrate how to use state machines to represent interesting computational processes.
Decomposition of state into variables, for convenience in representation and manipulation.
Deterministic vs. nondeterministic state machines.

Then introduce the key notion of "invariants":
Properties that are always preserved by state machines.
These are the main tool used in practice, in careful reasoning about algorithms and machines.

Show how invariants can be proved.
By induction.
Some that don't need strengthening of inductive hypothesis.
Others that do.
Proofs of invariants in batches.

Examples:
Numbers:
hopscotch Euclid's GCD algorithm
fast multiplication

Distributed algorithms:
Leader election
Alternating Bit Protocol

## 7.1 Definitions: State Machine, Executions and Reachability

*This section is a summary of a more complete version of Handout 11.*

A *state machine* is a triple $(Q, Q_0, \delta)$, where:

1. $Q$ is a nonempty (finite or infinite) set of *states*.
2. $Q_0 \subseteq Q$ is a nonempty subset of the states known as the *start states*.
3. $\delta$ is a set of *transitions* from state to state.

Don't confuse with 6.045 FSM's – these machines need not be finite.

Similar to digraphs, but has distinguished start states.

The point of a state machine is that it *executes*, producing *executions*.
Starts from any start state, and moves along, following transitions.
In this way, it models a computational process.

An *execution fragment* of a state machine is a finite or infinite sequence $q_0, q_1, \ldots$ such that $(q_i, q_{i+1}) \in \delta$ for every $i \geq 0$.

An *execution* is an execution fragment in which the initial state $q_0$ is an element of $Q_0$.

A state of $Q$ is said to be *reachable* if it is the final state of some finite execution.

Simple examples.
Begin with a simple state machine with one initial state, no choices of what to do next.
Has infinitely many states.

**Example 3** – **Deterministic** Counter

$Q = \mathbb{N}$, $Q_0 = \{0\}$

$\delta = \{(n, n+1) | n \in \mathbb{N}\}$

Intuitively, this models a counting process that starts with value 0 and increments (forever).

The reachable states are all of $Q = \mathbb{N}$.

Each natural number $n$ is only reached by an execution with $n$ transitions ($n+1$ states). There is only one execution of each number of transitions.

The counter is an example of a special kind of state machine that we call *deterministic.* This means that it has only a single start state: $|Q_0| = 1$,
and each state has at most one successor: for each $q \in Q$, there is at most one $q'$ such that $(q, q') \in \delta$.

If a state machine isn't deterministic, we say it's *nondeterminisitic.*
Means it has choices, either of starting state, transitions, or both.

The following is an example of a nondeterministic state machine:

**Example 4 – Non-deterministic** Linear hopscotch, with steps of 12 and 18:

$Q =$ integers
$Q_0 = \{0\}$

$\delta = \{(i, i+12) | i \in \mathbb{Z}\} \cup \{(i, i-12) | ...\} \cup \{(i, i+18) | ...\} \cup \{(i, i-18) | ...\}$.
Thas has many different executions, all leading to states that are multiples of the GCD, which is 6.

Can also vary this machine to allow multiple start states.
E.g., $Q_0 = \{i | i = 0 \pmod 6\}$.
E.g., $Q_0 = \{0, 2\}$.

In most examples of state machines that occur in practice, the state has *structure.*
Can describe it as containing different sorts of information, represented as values of *state variables.*
Different from graphs, which are usually thought of as unstructured.
(Normally represented by a dot for each vertex, arrow for each edge.)
The variable structure leads to concise, understandable representations.

**Example 5** 2-dimensional hopscotch, say with 12 and 18 in EW and 9 and 12 in NS direction.
$Q$: Consists of values for two *variables*:
*east-west* $\in \mathbb{Z}$, initially 0
*north-south* $\in \mathbb{Z}$, initially 0

Formally, $Q$ is the Cartesian product, $\mathbb{Z} \times \mathbb{Z}$.
And $Q_0$ consists of the single pair, $(0, 0)$.

$\delta$:
All transitions of the following forms:

$move\text{-}east(12)$
Effect:
$east\text{-}west := east\text{-}west + 12$

$move\text{-}east(18)$
Effect:
$east\text{-}west := east\text{-}west + 18$

$move\text{-}west(12)$
Effect:
$east\text{-}west := east\text{-}west - 12$

$move\text{-}west(18)$
Effect:
$east\text{-}west := east\text{-}west - 18$

$move\text{-}north(9)$
Effect:
$east\text{-}west := east\text{-}west + 9$

Etc.
The names we associate with the transitions are called *labels*.
They have no formal significance here, but are convenient for presenting the state machine.
Formally, $\delta$ consists of all pairs that are described by any of the little code fragments.
A code fragment describes a *set* of pairs of states, in an obvious way.
E.g., the first fragment denotes all pairs $(q, q')$, where *east-west* in $q'$ is 12 more than it is in $q$, and *north-south* is the same in $q'$ as it is in $q$.

Written concisely:
$q'.east\text{-}west = q.east\text{-}west + 12$,
$q'.north\text{-}south = q.north\text{-}south$.

The transitions above are described using assignment statements.
Nice for programmers.

Sometimes use equations (like the ones just above) instead.

One more wrinkle:
When using code fragments to represent state transitions, sometimes it's convenient to give a predicate, *precondition*, restricting when the code fragment is allowed to be applied.

**Example 6** 2-dim hopscotch on only northeast quadrant:

The same as above, only now the two state components *east-west* and *north-south* take on nonnegative values only (values in $\mathbb{N}$).

So now we have to be careful – some of the named transitions can't be applied in all situations, e.g.:

*move-west*(12)
Precondition:
*east-west* $\geq 12$
Effect:
*east-west* $:=$ *east-west* $- 12$

The precondition is a predicate on the state variables restricting when the transitions denoted by this code fragment are permitted to occur.
Formally, the transitions denoted by this fragment are all $(q, q')$ for which the precondition predicate is true of $q$ and for which $q'$ is obtained from $q$ by executing the effect code.

State machines can be used to describe most computing processes, including both algorithms and computing machines.
The state variable structure is usually crucial for obtaining understandable respresentations.

## 7.2 Invariants

*This section is a summary of a more complete version of Handout 11.*

Now we are ready to introduce the main practical method for reasoning about the correctness of algorithms and physical machines represented as state machines – the method of *invariants.*
Recall, we have proven correctness in recitation of an algorithm of fast multiplicatiion mod m and correctness of a state machine of fast multiplication. Recall, that in lecture 6 we also discussed how to prove corectness.

This method of invariants involves arguing that the machine always preserves properties of the state known as *invariants*. This is often the key to understanding what the machine does, and why.

**Definition 7.1** An *invariant* of a state machine $M$ is any predicate that is true of all the reachable states of $M$.

**Example 7** In the general 2-dim hopscotch, with initial state $(0, 0)$, the statements that
*east-west* is a multiple of 6,
and *north-south* is a multiple of 3,
are invariants.

Of course, these statements give a lot of insight into what this machine does.

How to prove that some state machine satisfies an invariant?
Must be true of all reachable states.
Reachable states are those that are reached by some number of transitions, beginning from a start state.
So, prove by induction on number of transitions.
Breaks down into base, 0 transitions, which shows the property is true in the initial states, and inductive step, which shows the property is preserved by all steps.

**Example 8** 2-dimensional hopscotch.

Invariant as stated above.
Prove by induction

BASE: Initial state $(0, 0)$ multiples of 6, 3 respectively.
INDUCTIVE STEP:
Suppose true for all states reachable after $n$ steps.
Now consider state reachable after $n + 1$ steps.
That state is reached by an execution with $n$ steps, then one more step.
Say the state after $n$ steps is $q$ and the final state is $q'$.
By inductive hypothesis, we may assume that the invariants are true for $q$.
That is, $q.\mathit{east\text{-}west}$ is a multiple of 6 and $q.\mathit{north\text{-}south}$ is a multiple of 3.
We must show that $q'.\mathit{east\text{-}west}$ is a multiple of 6 and $q'.\mathit{north\text{-}south}$ is a multiple of 3.

Consider cases based on type of transition.

(The usual, when the machine is represented by code fragments.)

Here, all the cases are basically the same, so I'll just do the first.
Since $q.\mathit{east\text{-}west}$ is a multiple of 6 and $q'.\mathit{east\text{-}west} = q.\mathit{east\text{-}west} + 12$, we also have $q'.\mathit{east\text{-}west}$ a multiple of 6.
Also (don't forget) since $q.\mathit{north\text{-}south}$ is a multiple of 3 and $q'.\mathit{north\text{-}south} = q.\mathit{north\text{-}south}$, we also have $q'.\mathit{north\text{-}south}$ a multiple of 3.

QED

## 7.3 Fast Multiplication

"Russian peasant's multiplication"

The states are composed of variables, all in $\mathbb{N}$:
$x$, $y$, the values to be multiplied, initially arbitrary positive
$a$, the accumulator for the answer
$r$, $s$, temporaries

Initially, $r = x$, $s = y$, $a = 0$.

Transitions:

*even*
Pre:
$s$ is even
$s > 0$
Eff:
$r := 2r$
$s := \frac{s}{2}$

*odd*
Pre:
$s$ is odd
Eff:
$a := a + r$
$r := 2r$
$s := \frac{s-1}{2}$

The answer is the value in the accumulator $a$ when $s = 0$.

This may not be completely obvious.
Example execution:

| $x$ | $y$ | $r$ | $s$ | $a$ |
|---|---|---|---|---|
| 5 | 9 | 5 | 9 | 0 |
| | | 10 | 4 | 5 |
| | | 20 | 2 | 5 |
| | | 40 | 1 | 5 |
| | | 80 | 0 | 45 |

Proof: To see this is correct, would like to prove invariant:

Invariant 1: If $s = 0$ then $a = xy$.

Of course, need to strengthen in order to prove.
Turns out that the key is:

Invariant 2: $xy = rs + a$

Notice how this expresses the idea that explains why the algorithm works.
This is typical for invariants.

So, can prove Invt1 and Invt2 together by induction, as before.
Alternatively, and slightly better, can just prove Invt2 by itself, by induction, and then infer Invt1 as a corollary.

Proving Invt2 is straightforward by induction:

BASE:
True initially because $r = x$, $s = y$ and $a = 0$.

INDUCTIVE STEP:
Consider cases based on type of action, *even* or *odd*.

*even*:
Doubling $r$ while halving $s$ doesn't change the product $rs$,
and $a$ doesn't change, so easy.

*odd*:

Assuming $(q.x)(q.y) = (q.r)(q.s) + q.a$.
Must show $(q'.x)(q'.y) = (q'.r)(q'.s) + q'.a$.

Now $q'.x = q.x$, $q'.y = q.y$, so LHS unchanged.
Also, $q'.r = 2(q.r)$, $q'.s = \frac{q.s-1}{2}$, and $q'.a = q.a + q.r$.
So $(q'.r)(q'.s) + q'.a = 2(q.r)(\frac{q.s-1}{2}) + q.a + q.r$.
Reorganizing terms yields $(q.r)(q.s) + q.a$, which suffices.

So Invt2 is proved.
For Invt1, don't even need an induction:

Already know that in all reachable states, $xy = rs + a$.
Need to show in all reachable states, if $s = 0$ then $a = xy$.

So let $q$ be any reachable state.
We know that $(q.x)(q.y) = (q.r)(q.s) + (q.a)$.
If $q.s = 0$, we get $(q.x)(q.y) = q.a$.
As needed.

This has illustrated the idea of strengthening an invariant, then proving only a part of it by induction and getting the rest as a consequence (corollary).

This strategy extends:
Sometimes to prove an invariant must extend it to a long list.
Then useful to prove one or a batch first, by induction.
After that, we can assume their results in proving another batch by induction.
Etc.
Anywhere along the way, might get some as consequences, without using induction.