Massachusetts Institute of Technology
6.042J/18.062J: Mathematics for Computer Science
Professors Nancy Lynch and Srini Devadas

Handout 21
October 2, 1998

Notes for Recitation 4 (10/2/98)

1 Greatest Common Divisor

In class this week, we studied invariants and how to use them. As another example of invariants, we will prove that Euclid's famous algorithm for computing the greatest common divisor of two numbers never gives the wrong answer. This is half of a proof that the algorithm is correct; the other half is to prove that the algorithm terminates. We do this by exhibiting a decreasing derived variable; a theorem presented in lecture then implies termination.

Definition 1.1 *The* greatest common divisor *(GCD) of two natural numbers x and y, not both zero, is the largest $n \in \mathcal{N}$ such that $n|x$ and $n|y$.*

For example, $GCD(6, 15) = 3$, $GCD(6, 7) = 1$, and $GCD(2, 0) = 2$. $GCD(0, 0)$ is undefined. Computing greatest common divisors is important in cryptography.

One way to compute the GCD of two numbers is to factor both into primes and then pull out prime factors shared by both. For example:

$$\begin{aligned} 24 &= 2^3 \cdot 3 \\ 36 &= 2^2 \cdot 3^2 \\ &\Rightarrow GCD = 2^2 \cdot 3 = 12 \end{aligned}$$

The problem is that factoring is very slow. Fortunately, the Greek mathematician Euclid thought of a better algorithm. (He apparently never implemented this algorithm; perhaps he was too lazy to learn C.)

The input to procedure `Euclid` is a pair of numbers (x, y). There are only two steps in the algorithm:

1. If $y = 0$, return x.
2. Otherwise, return `Euclid`$(y, x \bmod y)$.

(Recall that $x \bmod y$ denotes the remainder when x is divided by y.)

As an example, suppose that the procedure is called with inputs $x = 22$ and $y = 16$. This would generate a sequence of recursive calls: `Euclid`(22, 16), `Euclid`(16, 6), `Euclid`(6, 4), `Euclid`(4, 2), `Euclid`(2, 0). The final call returns 2. The value 2 propagates all the way back up the recursion and is output as the final answer. Sure enough, $GCD(22, 16) = 2$.

Here is the theorem that we want to prove about Euclid's algorithm:

Theorem 1.2 *If* `Euclid`(x, y) *terminates, then it returns* $GCD(x, y)$.

We can use the state machine/invariant approach to prove this theorem. The first problem is to define the three components of the state machine, Q, Q_0, and δ:

$$\begin{aligned} Q &= \{(a,b) \mid a,b \in \mathcal{R}\} \\ Q_0 &= \{(x,y)\} \\ \delta &= \{(a,b) \to (b, a \bmod b) \mid b \neq 0\} \end{aligned}$$

Two aspects of these definitions might be disturbing. First, the start state contains variables, x and y. This will not be a problem; we will treat x and y as constants in our analysis. The upshot is that we are defining a somewhat different state machine for each input.

Second, we have not defined a transition for the case when $b = 0$, even though this corresponds to step 1 of the algorithm. This is the termination condition, which we are deferring.

The only remaining preparatory work before the proof is choosing a predicate. Looking back at the example execution suggests the predicate $GCD(a,b) = GCD(x,y)$. That is, any stage the GCD of the remaining numbers is equal to the GCD of the original two numbers. Now we are ready to prove the theorem.

Proof. The proof uses the Invariant Theorem.

Let $P((a,b))$ be the predicate that $GCD(a,b) = GCD(x,y)$.

First, we must show that the predicate holds for all start states. This is trivial because the only start state is (x,y) and $GCD(x,y) = GCD(x,y)$.

Next, for each transition $(a,b) \to (a',b') \in \delta$, assume $GCD(a,b) = GCD(x,y)$ to prove $GCD(a',b') = GCD(x,y)$. It is sufficient to show that $GCD(a',b') = GCD(a,b)$. Futhermore, since all transitions in δ are of the form $(a,b) \to (b, a \bmod b)$, we have only to check that $GCD(b, a \bmod b) = GCD(a,b)$.

Let $s = GCD(a,b)$ and $t = GCD(b, a \bmod b)$. We will prove that $s = t$ by showing that each divides the other.

First, we show that s divides t. By definition of GCD, $s \mid a$ and $s \mid b$. For some integer c, $a \bmod b = a - bc$, and so $s \mid a \bmod b$. Since $s \mid b$ and $s \mid a \bmod b$, $s \mid t$.

Now, we show that t divides s. Again, by definition of GCD, $t \mid b$ and $t \mid (a - bc)$. These facts imply $t \mid a$, and so $t \mid GCD(a,b)$ and $t \mid s$. ■

This proves that at every stage the algorithm stores two numbers whose GCD is the same as the original input pair. If the second number ever becomes zero, then the correct answer will be returned by step 1 of the algorithm.

Now we address the question of whether the algorithm terminates, and if so how quickly it terminates. Recall from lecture the theorem about decreasing derived variables.

Theorem 1.3 *If $f : Q \to \mathcal{N}$ is a decreasing derived variable then the length of any execution starting at q_0 is at most $f(q_0)$.*

We will now use this to show that Euclid's algorithm terminates in at most x steps.

First let us establish a suitable invariant.

Claim 1.4 *If $x > y$ then $P((a, b)) : a > b$ is an invariant.*

Proof. To prove that P is an invariant, we must show that it hold for all start states and is preserved by all transitions. The invariant holds trivially in the start state, since by assumption $x > y$.

Next, we have to show that for each transition $(a, b) \to (a', b') \in \delta$, $P(a, b) \Rightarrow P(a', b')$. That is, we asum $a > b$ to prove $a' > b'$, where the algorithm specifies that $a' = b$ and $b' = a \bmod b$. This holds because $a \bmod b < b$ by definition of the mod operator. ∎

Let f be the derived variable $f((a, b)) = a$.

Claim 1.5 *If $x > y$ then f is a decreasing derived variable.*

Proof. Consider a transition from (a, b) to $(b, a \bmod b)$. In the state (a, b) we know from the invariant that $a > b$.

Therefore $f((a, b)) = a > b = f((b, a \bmod b))$. ∎

From the the second claim and the theorem proven in lecture, we conclude that if we start in a state where $x > y$ then the algorithm terminates in at most $f((x, y)) = x$ steps.

2 Fast Exponentiation

Exponentiation simply corresponds to computing r^n, where r is a real number, and n is a natural number. Of course, we can compute r^n using $n - 1$ multiplications. A faster algorithm is described below.

Fast Exponentiation: compute r^n for $r \in \mathbb{R}, n \in \mathbb{N}$.

State machine with states in $(a, b, k) \in \mathbb{R} \times \mathbb{R} \times \mathbb{N}$.

Initial state: $(1, r, n)$

Next state:
if $k > 0$ and k is even, then $(a, b, k) \to (a, b^2, k/2)$;
if $k > 0$ and k is odd, then $(a, b, k) \to (ab, b, k - 1)$.

Invariant: $ab^k = r^n$

Termination: k decreases.

3 15 Puzzle

In a 15 puzzle, square tiles numbered $1, \ldots, 15$ in a 4×4 array with one blank square. Any tile adjacent to the blank square can slide into it.

The state machine model of the puzzle has states consisting of a sequence (permutation), p, of $1, \ldots, 15$ (indicating the positions of the numbered tiles reading left-to-right and then top-to-bottom), with a nonnumerical token, b, inserted somewhere in the permutation to indicate where the blank square is. For example, the initial position

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	b

is represented by $1, 2, \ldots, 15, b$, and the target position

15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	b

is represented by $15, 14, \ldots, 1, b$.

Next state: Exchange the blank square and an adjacent tile. For example blank at square $(2, 2)$ exchanges position with tile above at $(1, 2)$:

n_1	n_2	n_3	n_4
n_5	b	n_6	n_7
n_8	n_9	$n_1 0$	$n_1 1$
$n_1 2$	$n_1 3$	$n_1 4$	$n_1 5$

$\longrightarrow$

n_1	b	n_3	n_4
n_5	n_2	n_6	n_7
n_8	n_9	$n_1 0$	$n_1 1$
$n_1 2$	$n_1 3$	$n_1 4$	$n_1 5$

corresponds to the state change

$$(n_1 n_2 n_3 n_4 n_5 \ b \ n_6 n_7 \ldots n_{15}) \to (n_1 \ b \ n_3 n_4 n_5 n_2 n_6 n_7 \ldots n_{15})$$

We prove that starting from the initial state, there is no way to reach the target state. For any permutation, p, of the integers $1, \ldots, n$, let inversions(p) be the number of out-of-order pairs there are in p. For example, with $n = 5$, the permutation 12453 has two out-of-order pairs: 43 and 53. The in-order sequence 12345 has no out-of-order pairs. More generally,

$$\text{inversions}(1, 2, \ldots, n) = 0 \text{ and inversions}(n, n-1, n-2, \ldots, 1) = n(n+1)/2.$$

Invariant: The parity of $k ::= \text{inversions}(p) + r$ where p is the permutation of $1, 2, \ldots, 15$ (ignoring the b), and r is the row number of the blank square. (Row one is at the top, and the blank is initially at the bottom, so initially $r = 4$.)

Notice that for the initial position, inversions$(1\ 2\ \ldots 15) = 0$ and $r = 4$, so k_{init} is even. For the target position, inversions$(15\ 14 \ldots 2\ 1) = 14 + 13 + \cdots + 1 = (15 \cdot 14)/2 = 105$ and

$r = 4$ so k_{target} is odd, so the invariant implies the target pattern is not reachable from the initial one.

To show that the value of k is constant, note that horizontal moves change nothing, and vertical moves both change r by 1, and move a tile three places forward or back in the current permutation, p. This reverses the order of three pairs in p, changing the number of inversions by 3 or 1, but always by an odd amount.

4 Extended GCD

Extended GCD may have been done in some recitations instead of the one of the previous ones. It is an interesting example of the invariant method, and it also establishes a very useful fact about gcd's: the gcd of two integers, a and b, can be expressed as a linear combination with integer coefficients of a and b.

Given integers a, b the following procedure returns the gcd of a and b, and also coefficients $k, l \in \mathbb{N}$ so $ka + lb = \gcd(a, b)$. (This procedure and its invariants are taken from Aho, Hopcroft, and Ullman's text on algorithms.)

```
u := 1; v := 0; k := 0; l := 1; x := a; y := b;
loop:
if y|x DONE;
q := floor(x/y);
** SIMULTANEOUS ASSIGNMENT:
 x := y,
 y := x - q * y   (This is the same as y =  x mod y )
 u := k,
 v := l,
 k := u - q * k,
 l := v - q * l;
goto loop;
```

NOTE: x, y behave exactly as in the standard Euclidean algorithm from lecture, except that this procedure stops one step sooner. This procedure terminates for the same reason as the Euclidean algorithm: y decreases (fast actually) each time around the loop.

The invariants we need to justify this procedure are:

1. $\gcd(x, y) = \gcd(a, b)$,

2. $ka + lb = y$, and

3. $ua + vb = x$.

The procedure terminates when $y|x$. Then invariants (a) and (b) imply $\gcd(a, b) = \gcd(x, y) = y = ka + lb$. So we have the gcd in register `y` and the desired coefficients in `k`, `l`. To verify the invariants:

Proof. Invariant (a) is the same one as for the Euclidean algorithm, which we proved in lecture. To check the other two invariants, note that at the start $k = 0, l = 1, y = b$ and $0a + 1b = b$ as required; also $u = 1, v = 0, x = a$ and $1a + 0b = a$ as required.

So we now assume all the invariants hold for u, v, k, l, x, y, and prove they hold for new values u', v', k', l', x', y'. Invariant (c) holds for $u' = k, v' = l, x' = y$ because of invariant (b) for k, l, y. Now $k' = u - qk$ and $l' = v - ql$, where $q = \lfloor x/y \rfloor$, so

$$k'a + l'b = (u - qk)a + (v - ql)b = ua + vb - q(ka + lb) = x - qy = y',$$

so invariant (b) holds for k', l'. ■

Team Problems

1. *Efficiency of Euclid's algorithm.*

   (a) Show that Euclid's GCD algorithm uses at most $2\log_2 x$ modulo operations. Assume that $x > y$. [Use derived variables]
   Derived variable: $f((a, b)) = a$

   Claim 4.1 *If $x > y > 0$ then for any pair of transitions, $q \to q_1 \to q_2$, $f(q_2) \leq \frac{1}{2}f(q)$, i.e. f halves every two steps.*

   Proof. We consider two cases:

   i. $f(q_1) \leq f(q)/2$.
   In this case, since f is a decreasing variable, $f(q_2) \leq f(q_1) \leq f(q)/2$.

   ii. $f(q_1) > f(q)/2$.
   Let $q = (a, b)$ with $a > b$. Then $q_1 = (b, a \bmod b)$ and $q_2 = (a \bmod b, b \bmod a \bmod b)$. Therefore $f(q) = a, f(q_1) = b$ and $f(q_2) = a \bmod b$. We know that $b > a/2$. Hence $a \bmod b < a/2$. That is $f(q_2) \leq f(q)/2$.

   ■

   (b) Let F_n denote the n^{th} Fibonacci number. Show that:

   i. If $2 \leq x \leq F_n$ then the algorithm uses at most $n - 2$ modulo operations. Assume $x > y$.
   By Strong Induction.
   I. H. If $2 \leq x \leq F_n$ then the algorithm uses at most $n-2$ modulo operations.
   Base. $x = 3$, $F_4 = 3$.
   Step. Assume $P(z)$ for all $z < x$. We will prove x.
   Suppose $2 \leq x \leq F_n$. Consider the next two transitions.

   $$(x, y) \to (y, x \bmod y) \to (x \bmod y, y \bmod (x \bmod y))$$

   Then $y + x \bmod y \leq x \leq F_n = F_{n-1} + F_{n-2}$. We consider two cases:

A. $y \leq F_{n-1}$.
Then the number of steps is at most 1 + number of steps from the state $(y, x \bmod y)$, which is at most $1 + n - 3$ by the induction hypothesis applied to y.

B. $y > F_{n-1}$. Therefore, $x \bmod y \leq F_{n-2}$.
The number of steps to termination is at most 2 + number of steps from the state $(x \bmod y, y \bmod (x \bmod y))$, which is at most $2 + n - 4 = n - 2$ steps by the I.H. applied to $x \bmod y$.

ii. For $n \geq 2$, $F_n \geq (\frac{3}{2})^{n-2}$.
By strong induction on n.
I.H. $P(n) : F_n \geq (\frac{3}{2})^{n-2}$.
Base: $n = 2$: $F_2 = 1 \geq (3/2)^0 = 1$.
Ind. Step: Assume $P(k)$ is true for $2 \leq k \leq n$. To prove $P(n+1)$.
$F_{n+1} = F_n + F_{n-1} \geq (\frac{3}{2})^{n-2} + (\frac{3}{2})^{n-3}$
$\geq (\frac{3}{2})^{n-3}(\frac{5}{2}) > (\frac{3}{2})^{n-3}(\frac{3}{2})^2$
$\geq (\frac{3}{2})^{n-1}$

iii. Using the previous two results show that Euclid's algorithm actually uses at most $1.71 \log_2 x$ modulo operations if $x \geq 2$.
Let k be the smallest integer such that $(\frac{3}{2})^k \leq x \leq (\frac{3}{2})^{k+1}$ Therefore $x \leq F_{k+3}$, and we know that Euclid's algorithm takes $k+1$ steps. But this is at most $\log_{3/2} x \leq 1.71 \log_2 x$.

■

2. *Choosy mating.*

Note that in this question, boys are assumed to be the aggressors, and they serenade the girls.

(a) We have the mating problem again, but now both boys and girls are more choosy about their partners. That is, among a set of n boys and n girls, each boy is unwilling to drop below his k^{th} choice and each girl is unwilliling to below her m^{th} choice. So it is possible that some of them are not married. How many pairs can we be sure of having in general ?

No, in general we cannot guarantee any marriages. E.g. Let $n = 10, k = m = 5$. $B_1, B_2, .., B_5$ rank only $G_1, \ldots, G_5$, $B_6, \ldots, B_{10}$ rank only $G_6, .., G_{10}$ and $G_1, \ldots, G_5$ rank only $B_6, \ldots, B_{10}$ and $G_6, \ldots, G_{10}$ rank only $B_1, \ldots, B_5$.

(b) What if only the boys were choosy, i.e., the girls are willing to marry anyone (they still have a ranked list of preferences), but the boys will not go below their k^{th} choice? How many pairs will the algorithm from class (TMA) assure us?

Set up a state machine as in the lecture. Each state is the set of ranking lists with names possibly crossed off. Each transition is one step of the algorithm, i.e. all boys who get a "no" answer cross off the girl who said "no" off their lists.

Recall also the derived variables from lecture.

$n(q)$ number of names not crossed off on all the boys lists put together.

$f_i(q)$ is the rank of girl that boy i is currently pursuing.

$g_i(q)$ is the rank of the boy that girl i said "maybe" to.

Termination. $n(q)$ is a decreasing variable. It starts at nk so, the algorithm terminates in nk steps at most.

At least k marriages. $f_i(q)$ is a non-decreasing variable while $g_i(q)$ is a non-increasing variable. If boy j remains unmarried then $f_j(q) = k + 1$. But this means that boy j serenaded all k girls on his list. So at least k girls have rejected him, because they have been serenaded by boys with higher rank (according to them). Consider any girl i in this set of k girls. Girl i has been serenaded by a boy p with $g_p(q) \leq n - 1$ (since girl i has rejected at least one boy). Further, boy p clearly has girl i within his top k choices, else he would not have serenaded her. Therefore girl i is definitely married at the end, either to boy p, or a higher-ranked boy. So k girls are all definitely married at the end.

No rogue couples. Identical to lecture.

Okay, so the TMA guarantees at least k marriages. Is this the best possible? Yes, consider e.g., $n = 10, k = 5$. For $i = 1$ to 5 B_i and G_i have each other at the top of their lists. And for $i = 6$ to 10, B_i has the list $G_1, ..G_5$. So $B_6, ..B_{10}$ will stay unmarried as all the girls on their lists have found better partners.