CSE 347 Lecture 1

Analysis of Algorithms – an example

1. Problem: Greatest Common Divisors
2. Analysis of correctness of Euclid's Algorithm
3. Analysis of running time of Euclid's Algorithm
4. Extensions & Applications

This course is about how to analyze algorithms – mainly, when should we be satisfied with an algorithm, and when should we look for a "better" algorithm.

Of course, this process usually doesn't start with an algorithm itself, but rather starts with a problem you want to be able to solve. Here is an example:

We say that an integer d divides an integer a if dividing a by d produces zero remainder (equiv. $a=cd$ for another integer c). For two integers a and b, the greatest common divisor of a and b, $GCD(a,b)$ is the largest d that divides both a and b.

For example, $GCD(35,100)=5$ since $35=5\cdot 7$ and $100=2^2\cdot 5^2$

How would you find the GCD? (discuss with neighbor)

Natural first guess – trial division. Note, this takes $\sim \min\{a,b\}$ divisions... OK for 35 & 100, but what about 12957 & 14917?

Notice that by comparison, adding or multiplying such large numbers is not a big deal. The algorithms we learned in grade school take a number of steps that grow linearly or quadratically in the # of digits and solve such tasks. It turns out that there is a similarly "good" algorithm for greatest common divisors:

Euclid's Algorithm (Elements, Book VII, Proposition 2):

Input: integers $a > b \geq 0$

Initialize $r \leftarrow a$, $r' \leftarrow b$

While $r' \neq 0$

  Put $r'' \leftarrow r \bmod r'$ ← (That is, the remainder of

  Put $(r, r') \leftarrow (r', r'')$ dividing r by r')

Output r.

This is history's first "nontrivial" algorithm, meaning

1. It is not obvious why it works ("correctness")
2. It is not obvious that it is fast

But it is fast. Let's compute GCD(14191, 12957):

$14191 = 1 \cdot 12957 + 1234$

$12957 = 10 \cdot 1234 + 617$

$1234 = 2 \cdot 617 + 0$

So GCD(14191, 12957) = 617 (Verify: $14191 = 23 \cdot 617$, $12957 = 2 \cdot 3 \cdot 617$ ✓)

(cf. how long would trial division take?)

Correctness of Euclid's Algorithm

Intuitively, if d divides a and b, d also divides a-b; if $a > 2b$, also d divides $a-2b$, ... and in general, if $a = cb + r$, then d divides $a - cb = r$, the remainder of dividing a by b. So, r has all of the common divisors of a and b. Now, suppose d divides b and r (the new r, r') — since $b = c' \cdot d$, $r = c'' \cdot d$, where also $r = a - c \cdot b$, $a - c \cdot b = c'' \cdot d$

$$a - c \cdot \overbrace{c' \cdot d} = c'' \cdot d \Rightarrow a = [c'' + c \cdot c'] \cdot d$$

So d also must have divided a and b — the pairs (a,b) and (b,r) have the same divisors

Claim in the ith iteration of Euclid's Algorithm, r, r' have the same divisors as a, b

Proof by induction on i. Base: $(a,b) = (r,r')$, trivial ✓
Induction step — the above argument.

We also obtain, by the properties of division, that in every iteration, $0 \le r'' < r' < r$, so (r, r') decrease on every iteration, hence eventually $r'' = 0$ and the algorithm stops. Observe that at this iteration, $r = c \cdot r' + 0$, so $r' = GCD(r, r') = GCD(a, b)$, which is what we return. (Correctness)

Running time (Discuss if time...)

In order to say why the algorithm is "fast," we need to somehow measure how long it takes. This is trickier than it first appears— should we assume we use the

grade-school algorithm for division? Or, that we have a calculator, and computing division is instantaneous? The "# of steps" changes dramatically depending on how we choose to measure it. You might think that coding it up and measuring the execution time is the "obvious" way to do this — and this is what early research in algorithms did. Unfortunately they discovered that when the underlying machine changed, the running time curve's shape changed, not just its scale.

In deference to these issues, we'll mostly ignore constant factors — we'll use the "big-O" notation: Recall, we say $f(n) = O(g(n))$ if, for sufficiently large n, there exists a constant C such that $f(n) \leq C \cdot g(n)$. We'll analyze the running time of our algorithms on an idealized RAM machine model in which loading, storing, and basic arithmetic can each be done in some unspecified "O(1)" steps. (further precision isn't worth the trouble!)

Today, we'll only consider the worst-case running time: for each input size (in digits), we'll bound the number of steps taken as a function of this size. (We will also consider some alternative measures of performance later.)

Theorem (Lamé, 1844): Euclid's Algorithm runs for $\leq \frac{\log b}{\log \phi} + 1$

iterations, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. ($\phi^2 = \phi + 1$)

Proof: We argue by induction on i, that when the algorithm has i iterations remaining, that $r' \geq \phi^i$ (hence, if the algorithm takes ℓ iterations, $b \geq \phi^{\ell-1} \Rightarrow \ell \leq \frac{\log b}{\log \phi} + 1$ ✓)

For the base cases, when $i=0$, $r' > 0$ still, so $r' \geq 1 = \phi^0$; and, when $i=1$, $r' > r'' \geq 1$, so $r' \geq 2 > \phi^1$.

Now, for the induction step, we note that $r_i \geq r' + r''$ where by the IH, $r' \geq \phi^{i-1}$ and $r'' \geq \phi^{i-2}$, so $r \geq \phi^{i-1} + \phi^{i-2} = (\phi+1)\phi^{i-2}$

Using $\phi + 1 = \phi^2$, we find $r \geq \phi^2 \phi^{i-2} = \phi^i$ ✓ ▨

Since the integer b has $\log b$ digits, this gives a $O(n)$ running time bound for n-digit inputs in our model.

(If we counted the cost of grade-school division, it turns out that the algorithm still only takes $O(n^2)$ steps.)

Extended Euclidean Algorithm

If we record the results of the division at each stage, we obtain a system of equations: for $a = r_0, b = r_1$.

$r_0 = q_1 r_1 + r_2$, $r_1 = q_2 r_2 + r_3$, $\cdots$, $r_{i-1} = q_i r_i + r_{i+1}$, $\cdots$, $r_{\ell-1} = q_\ell r_\ell$

That is: $\begin{bmatrix} r_i \\ r_{i+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix} \begin{bmatrix} r_{i-1} \\ r_i \end{bmatrix}$. Observe that iterating, we obtain $\begin{bmatrix} r_\ell \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_\ell \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_{\ell-1} \end{bmatrix} \cdots \begin{bmatrix} 0 & 1 \\ 1 & -q_1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$: so we can find integers s, t such that $s \cdot a + t \cdot b = d$ ($= GCD(a,b)$) by considering the top row of the matrix product (in time $O(n)$.) This new algorithm is the Extended Euclidean Algorithm.

This algorithm turns out to be useful for modular arithmetic-

Theorem The Extended Euclidean Algorithm, on input

any prime p and nonzero a<p returns (s,t) such that $t \cdot a \bmod p = 1$.

Proof: Since p is a prime number, GCD(p,a)=1. Therefore, the algorithm returns (s,t) such that $s \cdot p + t \cdot a = 1$. Now, notice $(s \cdot p + t \cdot a) \bmod p = t \cdot a \bmod p$ and $1 \bmod p = 1$ ▨

So this tells us how to "divide" in arithmetic modulo a prime. Actually, for expressions involving addition and multiplication "mod p", we can freely apply "mod p" to intermediate results – note that if $a = q_1 p + r_1$ and $b = q_2 p + r_2$, then $(r_1 + r_2) \bmod p = ((q_1 + q_2)p + r_1 + r_2) \bmod p = (a+b) \bmod p$ and $r_1 \cdot r_2 \bmod p = ((q_1 q_2 p + q_1 r_2 + q_2 r_1)p + r_1 r_2) \bmod p = a \cdot b \bmod p$

So, in particular, we can now solve systems of linear equations modulo a prime using only integer arithmetic. If the system $A\vec{x} = \vec{b}$ has a unique (integer) solution $\vec{x}$ that is less than p in each entry, then $\vec{x} \bmod p = \vec{x}$, and hence the solution $\vec{x}$ to $(A\vec{x} - b) \bmod p = 0$ is also the solution to $A\vec{x} - b = 0$ — but solving modulo p saves us from rounding errors due to finite precision arithmetic. We'll also see later in the course that the ability to solve linear systems of equations modulo a prime number will help us analyze hash functions.