Handout for recitation 12

21-127 sections A and F — TA: Clive Newstead — 25^{th} February 2014

Computational complexity of the Euclidean algorithm

In Monday's lecture you saw the Euclidean algorithm, a step-by-step process which, given $a, b \in \mathbb{Z}$ with $0 < |a| \leqslant |b|$, is guaranteed to output their greatest common divisor, denoted $\gcd(a, b)$.

The real question is: **why did we bother?** There's a much more 'straightforward' naïve procedure finding the greatest common divisor of a and b, as follows: for each integer from 1 up to $|a|$, check if it divides both a and b; if it does, write it down (otherwise forget it). The largest number you wrote down is $\gcd(a, b)$.

The reason why we bothered to show you the Euclidean algorithm is: using the above straightforward naïve method is extremely inefficient. Calculating greatest common divisors of large numbers is a common real-world task, especially in the computer security industry. (Read about the *RSA cryptosystem*—it's what makes your online bank transfers secure, amongst many other things.) If we had to go through the process of checking every natural number $\leqslant |a|$, our computers would burn out before they finished their task. We need a more time-efficient way.

The Euclidean algorithm is, in a very precise way, more efficient. Instead of requiring roughly[1] $|a|$ steps, as in the naïve approach, the Euclidean algorithm requires only roughly $\log(|a|)$ steps. If you don't know what logarithms are yet, don't worry; all you need to know is that, as $|a|$ gets larger and larger, $\log(|a|)$ grows much more slowly than $|a|$.

The essence of the vague waffle written above follows immediately from the following precise theorem:

Theorem 1. Let $a, b \in \mathbb{Z}$ with $0 < a \leqslant b$. Let $q_1, q_2, r_1, r_2 \in \mathbb{Z}$ be such that $0 \leqslant r_1 < a$, $0 \leqslant r_2 < b$ and

$$b = q_1 a + r_1 \qquad \text{and} \qquad a = q_2 r_1 + r_2$$

Then $r_1 + r_2 \leqslant \frac{3}{4}(a + b)$.

Proof. We know that $a \leqslant b$, so either $b \geqslant 2a$ or $a \leqslant b < 2a$. We consider these cases separately.

[*Case 1.*] Suppose $b \geqslant 2a$.

Since $r_1 < a$ and $r_2 < r_1$ we have $r_2 < a$ by transitivity. Since $b \geqslant 2a$, it follows that $a \leqslant \frac{1}{2}b$, and

[1] By 'roughly' I mean 'to the order of'. There is a precise way of measuring the complexity of an algorithm, which is not covered in 21-127, but leads to a very rich, interesting, useful, beautiful and applicable theory called *(computational) complexity theory*. Study it if you ever want to be employed.

since $r_1 < a$ we have $r_1 < \frac{1}{2}b$. In summary

$$\begin{aligned} r_1 + r_2 &< \frac{1}{2}b + a && \text{since } r_1 < \frac{1}{2}b \text{ and } r_2 < a \\ &= (\frac{3}{4} - \frac{1}{4})b + (\frac{3}{4} + \frac{1}{4})a && \text{re-writing} \\ &= \frac{3}{4}(a+b) - \frac{1}{4}(b-a) && \text{distributing and factorising} \\ &\leqslant \frac{3}{4}(a+b) && \text{since } b - a \geqslant 0 \end{aligned}$$

So by transitivity again, $r_1 + r_2 \leqslant \frac{3}{4}(a+b)$. So the result holds in Case 1.

[*Case 2.*] Suppose $a \leqslant b < 2a$.

First we prove that $q_1 = 1$. We do this by deriving contradictions from $q_1 \leqslant 0$ and $q_1 \geqslant 2$.

- If $q_1 \leqslant 0$ then $q_1 a \leqslant 0$, so $q_1 a + r_1 \leqslant r_1$. But $b = q_1 a + r_1$, so $b \leqslant r_1$. Since $a \leqslant b$, it follows from transitivity that $a \leqslant r_1$, contradicting the fact that $r_1 < a$.
- If $q_1 \geqslant 2$ then $q_1 a \geqslant 2a$, so $q_1 a + r_1 \geqslant 2a + r_1$. But $b = q_1 a + r_1$, so $b = 2a + r_1$. Since $b < 2a$, $b > b + r_1$, so $0 > r_1$, contradicting the fact that $r_1 \geqslant 0$.

The only remaining possibility is that $q_1 = 1$, and hence $b = a + r_1$. So $r_1 = b - a$. Since $b < 2a$ it follows that $a > \frac{1}{2}b$, so

$$r_1 = b - a < b - \frac{1}{2}b = \frac{1}{2}b$$

That is, $r_1 < \frac{1}{2}b$. Since also $r_2 < a$, we now know that $r_1 + r_2 < \frac{1}{2}b + a$. The same chain of inequalities as in Case 1 thus applies, leading to the conclusion that $r_1 + r_2 \leqslant \frac{3}{4}(a+b)$. So the result holds in Case 2.

Since all possible cases are covered, the theorem is proved. □

This leads us to the conclusion that, every time we take two steps in the Euclidean algorithm, the sum of the numbers that we're dealing with decreases by a factor of $\frac{3}{4}$. Hence the Euclidean algorithm certainly terminates in $2k$ steps, where k is the smallest natural number such that

$$\left(\frac{3}{4}\right)^k (a+b) < 1$$

After some easy logarithm calculations (not expected from you in 21-127), it follows that the Euclidean algorithm terminates in at most $C\log(|a|)$ steps, for some constant value C. This is, in general, much quicker than $|a|$ steps, especially when $|a|$ grows very large.

You may have noticed that Theorem 1 only applies when a and b are positive. For the negative case, the same result about complexity will follow once you've proved that, no matter what the input (i.e. the values of a and b), after two iterations of the Euclidean algorithm, all the numbers that appear in the rest of the algorithm are non-negative. I leave this bit to you. (**Hint:** by definition of remainders, $r_1 \geqslant 0$ and $r_2 \geqslant 0$.)