Time for Euclidean Algorithm

Suppose a, b have n digits and we apply $EUCLID(a, b)$. How long does it take. We know there are $O(n)$ iterations but each iteration, being a division (that is, finding a mod b for some a, b) might take $O(n^2)$ so it looks like $O(n^3)$. But, actually, it is $O(n^2)$.

To see why lets write a, b in binary, for example, $a = 111000010101$ and $b = 100010101$. We start the division

```
                 1
          ------------
100010101|111000010101
          100010101
          ---------
           10101101
```

How long does this step take? Well, there is a subtraction which takes $O(n)$ steps. (In any intermediate stage the numbers have at most n digits.) There is placing the 1 above, but that can go in one of two places. If you try one place and it fails (because the subtraction yields a negative number) it took just time $O(n)$ to see the failure. So this whole step takes time $O(n)$.

We continue the division to the end:

```
                 111
          ------------
100010101|111000010101
          100010101
          ---------
           1010110111
           100010101
           ----------
             100011011
             100010101
             ---------
                   100
```

In this case we were pretty lucky and got a very small $r = 100$. But the key observation is: Every digit we put up in the quotient knocks at least one digit off of the a value. Let us define `size` as the number of digits of a

plus the number of digits of b, where by a, b we here mean the current pair of numbers where we are applying the division. If we get an s-digit quotient it takes time $O(sn)$ but the new remainder has s fewer (maybe even better!) digits than a and so the cost (because on the next iteration we deal with b, r) has gone down by s. Say a, b are initially n-digit numbers so the cost is $2n$. At the end the cost can't be negative so it has gone down by at most $2n$. It is costing *time* $O(n)$ (the subtraction) for each reduction in the `size` by one. Thus the total time is $O(n^2)$.

Here is a slightly different approach, slightly modifying the Algorithm, with the same answer. The input is a with s digits and b with t digits with $s \geq t$. We define a *subtraction step* by setting

$$a' = a - 2^{t-s}b$$

This takes $O(n)$ steps. (The multiplying by 2^{t-s} simply moves the array for b over $t - s$ places, its not a true multiplication in terms of time.) Now, as argued before with $a = bq + r$, we have

$$\gcd(a, b) = \gcd(a'b)$$

With (the example above) we have $a = 111000010101$ and $b = 100010101$.

```
    a     111000010101
 2^4b     100101010000
    a'    010011000101
```

It may be that $a' < 0$. If that happens replace it by $|a'|$.

Now a' has (at least) one less digit before so that the *size* (the number of digits in a plus the number of digits in b) has gone down by (at least) one. We check which of a', b now has more digits and reverse them if necessary. If $a' = 0$ we end and return b as the gcd.

As the initial size was at most $2n$ (by assumption), we do at most $2n$ subtraction steps, each takes time $O(n)$ so the total time is $O(n^2)$.