Prof. Friedrich Eisenbrand
Alfonso Cevallos

Discussions from: March 3, 2015

Computer Algebra

Spring 2015

Assignment Sheet 2

Note: These are just notes and not necessarily full solutions to each exercise. Please report any mistakes you may find.

Exercise 1

Correctness: We denote by (a,b) the greatest common divisor of a and b. The details are left to you, but to prove that the algorithm actually outputs (a,b) basically we have to prove the following cases, all of which are straight-forward:
a) if a and b are even, then $(a,b) = 2 * (a/2, b/2)$;
b) if a is even and b is odd, then $(a,b) = (a/2, b)$ (similarly in the opposite case); and
c) if a and b are odd, then $(a,b) = (a-b, b)$ (actually this is true for any a, b).

Running time: For input a, b, the input size is $n = size(a) + size(b)$. The variables x, y start with values a, b, respectively, and every time one of them is divided by 2, they lose a bit; hence there can be at most n divisions by 2 in total, over all the loops. Similarly, there can be at most n addition and subtraction operations in total. This is because every addition is preceded by at least one division by 2, and every subtraction is followed by at least one division by 2. As there are $O(n)$ operations in total, each taking linear time at most, we obtain a bound of $O(n^2)$ on the running time. This estimation is tight: consider the instance $a = 2^n - 1,\ b = 1$. Besides any other operation, the algorithm performs the sequence of subtractions $\text{Sub}(2^n - 1,\ 1)$, $\text{Sub}(2^{n-1},\ 1)$, $\text{Sub}(2^{n-2} - 1,\ 1)$, etc. Just writing down the results of these operations takes time $\Omega(n^2)$.

Exercise 2
Recall the Extended Euclidean Algorithm $exgcd(a,b)$:

Data: $a, b \in \mathbb{N}$ with $a \geq b,\ a > 0$
Result: (g,x,y) such that $g = gcd(a,b) = ax + by$
if $b == 0$ **then**
  return $(a,1,0)$;
else
  $(q,r) = div(a,b)$; #(division with remainder)
  $(g,x',y') = exgcd(b,r)$;
  return $(g, y', x' - q * y')$
end

We prove that for $a \geq b > 0$, we have $|x| \leq b/g$ and $|y| \leq a/g$, by induction on m = number of recursive calls of the algorithm.
If $m = 0$, it must be the case that $b = 0$, so we don't need to prove anything.
If $m = 1$, then $r = 0$ and b divides a. We get $g = b$, $x = 0$ and $y = 1$; and the claim follows.
If $m > 1$, our induction hypothesis is $|x'| \leq r/g$ and $|y'| \leq b/g$; then $|x| = |y'| \leq b/g$ and $|y| = |x' - qy'| \leq |x'| + q|y'| \leq \frac{bq+r}{g} = a/g$.

Similarly to what we did in class, we prove that there is a constant C such that $exgcd(a,b)$ takes at most $C\log(a+1)\log(b+1)$ bit operations. The relevant operations are the division with remainder, the recursive call, and the multiplication. Assume we use simple multiplication, and notice that its input values are q and y', with $|y'| = |x| \leq b/g \leq b$. Hence we can select the constant C to be large enough so that the multiplication + division with remainder takes $\leq C\log(q+1)\log(b+1)$ bit operations. This, added to $\leq C\log(b+1)\log(r+1)$ bit operations for the recursive call, gives at most $C\log(b+1)[\log(q+1)+\log(r+1)] = C\log(b+1)\log[q(r+1)+r+1] \leq C\log(b+1)\log(qb+r+1) \leq C\log(b+1)\log(a+1)$ bit operations.

Exercise 3

See the Python code.

Exercise 4

Algorithm. Start by computing $N = \prod_{i=1}^{t} N_i$. Then, for each $i = 1, \cdots, t$, let $M_i = N/N_i$, compute $(1, x_i, y_i) = exgcd(N_i, M_i)$, and define $b_i = a_i * y_i * M_i \mod N$. Output $a = \sum_{i=1}^{t} b_i \mod N$.

Correctness. For a fixed i, the numbers N_i and M_i are relatively prime, hence we indeed have $1 = x_i * N_i + y_i * M_i$. It is easy to see that the definition of the number b_i gives $b_i \equiv a_i \mod N_i$, but $b_i \equiv 0 \mod M_i$, which implies that $b_i \equiv 0 \mod M_j$ for all $j \neq i$. Then it follows that the number we are looking for is simply the sum of the b_i's.

Complexity analysis. Let's define $s_i := size(N_i)$, and $S := size(N)$. Assuming that $N_i \geq 2$ for each i, we can prove that $\frac{1}{2}\sum_i s_i \leq S \leq \sum_i s_i$, which implies that $S = \Theta(\sum_i s_i)$. It is straightforwad to prove that the i-th iteration of the algorithm is executed in time $O(S * s_i)$ (for this, remember that $|y_i| \leq N_i$). So all iterations run in time $O(S\sum s_i) = O(S^2)$. The computations done before and after the t iterations can also be performed in this time complexity.

Exercise 5

See the Python code (we assumed the input n is given in its bit representation). Fib_1 is trivially correct. To prove correctness of Fib_2, it is enough to prove that if $A = \binom{1\ 1}{1\ 0}$, then $A^n = \binom{F_{n+1}\ F_n}{F_n\ F_{n-1}}$. This is easily done by induction on n.

We proved in class that Fib_1 takes time exponential in n. We studied in the same lecture another algorithm, that computes all the first n Fibonacci numbers in sequence, and thus its running time is $\Theta(n^2)$ (recall that the n-th Fibonacci number has size $\Theta(n)$). The performance of Fib_2 may be worse or better than that, depending on its implementation. To prove this point, we provide 3 different implementations in Python, and analyse each of them here.

Let M_n be the complexity of multiplying 2 numbers of size n. If we use Karatsuba, then $M_n = \Theta(n^{log_2 3}) = \Theta(n^{1.585})$. In $Fib2a$ we implement the "slow" matrix exponentiation, where we multiply A by itself n times. Since every matrix multiplication requires a constant number of integer multiplications, we are performing a total of $O(n)$ multiplications, for numbers of size $O(n)$. Hence the total complexity is $O(nM_n)$ (worse than quadratic).

In $Fib2b$, we use the "fast" matrix exponentiation, where we square the matrix over and over, $size(n)$ times. By doing so, we only perform $O(\log n)$ integer multiplications. However, unlike in the *modular exponentiation* case, the sizes of the numbers being multiplied grow exponentially fast. It turns out that the time complexity of the whole algorithm is asymptotically equivalent to the complexity of the largest multiplication, which is between two numbers of size $O(n)$. Therefore the complexity is $O(M_n)$ (better than quadratic).

Finally, $Fib2c$ is the same as $Fib2b$, but we optimize the matrix squaring operation to our particular use. Since any power of A is of the form $\binom{p+q\ p}{p\ q}$, we can square it using only 2 integer multiplications (see the code); in contrast, a standard matrix multiplication takes 8 integer multiplications. As such, $Fib2c$ has the same complexity as $Fib2b$, but is faster by a constant factor. With our implementation, Fib_1 can only compute up to the 2^5-th Fibonacci number befor crashing. $Fib2a$ starts being much slower than $Fib2b$ for $n = 2^9$. Finally, $Fib2c$ is still reasonably fast for $n = 2^{13}$, and more than 4 times faster than $Fib2b$.

Exercise 6

Let $g = gcd(a,b,c)$ and $d = \min\{xa+by+cz \mid x,y,z \in \mathbb{Z},\ xa+by+cz > 0\}$. Any common divisor of a,b and c also divides any integer combination of them; so in particular $g|d$, and so $g \leq d$. On the other hand, we claim that d is a common divisor of a,b and c, which implies $d \leq g$, and hence $d = g$. Assume by contradiction and without loss of generality that d does not divide a. Then $a = qd + r$, for some integers q,r with $0 < r < d$; but $r = a - qd$ is also an integer combination of a,b,c, which is a contradiction on the minimality of d.

Exercise 7

Suppose $a \cdot b = e$. Then, $e = (b \cdot a)^{-1} \cdot (b \cdot a) = (b \cdot a)^{-1} \cdot b \cdot e \cdot a = (b \cdot a)^{-1} \cdot b \cdot a \cdot b \cdot a = b \cdot a$.

If $e = a \cdot b \cdot c = (a \cdot b) \cdot c$, then $e = c \cdot (a \cdot b)$, from the proof above. However, it does not follow that $a \cdot c \cdot b = e$. Consider the group of invertible $2x2$ matrices with multiplication. If $a = \binom{1\ 0}{2\ 1}$, $b = \binom{1\ 1}{0\ 1}$ and $c = \binom{3\ -1}{-2\ 1}$, then $abc = \binom{1\ 0}{0\ 1}$ but $acb = \binom{3\ 2}{4\ 3}$.

Exercise 8

Let $G = a\mathbb{Z} + b\mathbb{Z}$ and $H = a\mathbb{Z} + (5a+b)\mathbb{Z}$. If $s = ax + by$ and $t = ax' + by'$ are elements in G, their difference $s - t = a(x - x') + b(y - y')$ is clearly also in G, hence G is a subgroup of $\mathbb{Z}$ (remember the characterization of subgroup seen in class). Now, element $s = ax + by = a(x - 5y) + (5a + b)y$ is clearly also in H, which proves that $G \subset H$; and in a similar way we can prove that $H \subset G$, thus $G = H$.

Finally, we proved in class that any non-trivial subgroup of $\mathbb{Z}$ looks like $d\mathbb{Z} = \{dx \mid x \in \mathbb{Z}\}$, where d is the smallest positive number in the subgroup. The subgroup G consists of all the integer combinations of a and b, and from Bezout's lemma we know that $g = gcd(a,b)$ is precisely the smallest positive number in such set. Hence $G = H = g\mathbb{Z}$. (Note: we assumed here that a and b are not both zero; the case $a = b = 0$ should be treated separately.)

Exercise 9

Suppose $d = gcd(n_1, n_2)$, and let a_1, a_2 be two given integers. We prove that

$$\exists \text{ integer } a \text{ with } a \equiv a_i \mod n_i \Leftrightarrow a_i \equiv a_2 \mod d$$

Forward implication. If there is such an integer a, then $a \equiv a_1 \mod n_1$ implies $a \equiv a_1 \mod d$ (because $d|n_1|a - a_1$); and similarly we get $a \equiv a_2 \mod d$. Thus $a_1 \equiv a_2 \mod d$.

Backward implication. For a prime p and an integer n, we define $ord_p(n)$ as the exponent of the largest power of p that divides n, i.e. $p^{ord_p(n)}|n$, but $p^{1+ord_p(n)} \nmid n$. It will be useful to notice that $d = gcd(n_1, n_2) = \prod_p p^{\min\{ord_p(n_1), ord_p(n_2)\}}$, and similarly $lcm(n_1, n_2) = \prod_p p^{\max\{ord_p(n_1), ord_p(n_2)\}}$, where the product is over all primes p that divide n_1 or n_2, and $lcm(n_1, n_2)$ is the least commom multiple of n_1 and n_2.

The system of modular equations $a \equiv a_i \mod n_i$, $i = 1,2$, can be rewritten as the extended system $a \equiv a_i \mod p^{ord_p(n_i)}$, over $i = 1,2$ and primes p. The fact that $a_1 \equiv a_2 \mod d$ ensures that, for every prime p, the two equations modulo a power of p are coherent, and no information is lost if we keep the one with the highest power, and discard the other one. The resulting system has pairwise coprime moduli, so the Chinese remainder theorem ensures that it has a solution a. (This solution will be unique modulo $\prod_p p^{\max\{ord_p(n_1), ord_p(n_2)\}} = lcm(n_1, n_2)$).