13. The most time-consuming parts of implementing the Chinese Remainder Theorem are: (i) computing M; (ii) computing $M_i = M/m_i$ for each of the r different i's; (iii) finding the inverse of M_i modulo m_i for each i; (iv) multiplying out $a_i M_i N_i$ in the formula for x for each i; (v) dividing the resulting x by M to get the least nonnegative value. We use $O(\log B)$ for the number of bits in the m_i or a_i or N_i, and $O(r \log B)$ for the number of bits in M or the M_i. This gives $O(r^2 \log^2 B)$ for the number of bit operations to do (i)–(ii), (iv)–(v). In (iii), we need $O(r^2 \log^2 B)$ bit operations to reduce each of the M_i modulo the corresponding m_i before taking the inverses, and then $O(r \log^3 B)$ bit operations to find all r inverses by the Euclidean algorithm. This gives the combined estimate $O\big(r \log^2 B(r + \log B)\big)$. Whether the $r^2 \log^2 B$ term or the $r \log^3 B$ term dominates depends on the relative size of r and $\log B$ (i.e., the number of equations and the number of bits in our moduli).
14. $38^{1+2+2^3+2^6} \equiv 38 \cdot 2 \cdot 16 \cdot 63 \equiv 79 \ mod\ 103$.
15. If we use the $O(k^2)$ estimate for the time to perform one multiplication of k-bit integers (as we have been doing), then there is no saving of time. In fact, the very last multiplication already uses time $O((n \log b)^2)$, which is the estimate we get by multiplying b by itself n times. The difficulty is that, unlike in modular arithmetic, in the repeated squaring method we end up dealing with pairs of very large integers, and this offsets the advantage of having far fewer multiplications to perform. But if we were to use a more clever way of multiplying two k-bit integers, for example, if we used an algorithm requiring only $O(k \log k \log\log k)$ bit operations, then it would save time to use the repeated squaring method.
16. (a) Repeated squaring requires $O(\log^3 p)$ bit operations whereas a time estimate of $O(\log^2 p)$ can be proved for the Euclidean algorithm. (b) Repeated squaring still requires time $O(\log^3 p)$, but after we perform the first step of the Euclidean algorithm — dividing p by a (which requires $O(\log p \log a)$ bit operations) — the rest of the Euclidean algorithm takes $O(\log^2 a)$ bit operations. So the Euclidean algorithm is faster, especially for a very small compared to p.
17.

n	90	91	92	93	94	95	96	97	98	99	100
$\varphi(n)$	24	72	44	60	46	72	32	96	42	60	40

18. There is no n for which $\varphi(n)$ is an odd number greater than 1; $\varphi(n) = 1$ for $n = 1, 2$; $\varphi(n) = 2$ for $n = 3, 4, 6$; $\varphi(n) = 4$ for $n = 5, 8, 10, 12$; $\varphi(n) = 6$ for $n = 7, 9, 14, 18$; $\varphi(n) = 8$ for $n = 15, 16, 20, 24, 30$; $\varphi(n) = 10$ for $n = 11, 22$; $\varphi(n) = 12$ for $n = 13, 21, 26, 28, 36, 42$. To prove, for example, that these are all of the n for which $\varphi(n) = 12$, compare the possible factorizations of 12 (with 1 allowed as a factor but not 3) with the formula $\varphi(\prod p^\alpha) = \prod(p^\alpha - p^{\alpha-1})$. One has $1 \cdot 2 \cdot 6$, $1 \cdot 12$, $2 \cdot 6$, and 12. The first gives $2 \cdot 3 \cdot 7$, the second gives $2 \cdot 13$, the third gives $(3 \ or\ 4) \cdot 7$ and $4 \cdot 9$, and the fourth gives 13.