Primality tests - Algorithms for Competitive Programming

Source: https://cp-algorithms.com/algebra/primality_tests.html

Last update: April 16, 2024 Original Primality tests ¶ This article describes multiple algorithms to determine if a number is prime or not. Trial division ¶ By definition a prime number doesn't have any divisors other than 1 and itself. A composite number has at least one additional divisor, let's call it d . Naturally $\frac{n}{d}$ is also a divisor of n . It's easy to see, that either $d \le \sqrt{n}$ or $\frac{n}{d} \le \sqrt{n}$, therefore one of the divisors d and $\frac{n}{d}$ is $\le \sqrt{n}$. We can use this information to check for primality. We try to find a non-trivial divisor, by checking if any of the numbers between 2 and $\sqrt{n}$ is a divisor of n . If it is a divisor, then n is definitely not prime, otherwise it is. bool isPrime (int x) { for (int d = 2 ; d * d <= x ; d ++) { if (x % d == 0) return false ; } return x >= 2 ; } This is the simplest form of a prime check. You can optimize this function quite a bit, for instance by only checking all odd numbers in the loop, since the only even prime number is 2. Multiple such optimizations are described in the article about integer factorization . Fermat primality test ¶ This is a probabilistic test. Fermat's little theorem (see also Euler's totient function) states, that for a prime number p and a coprime integer a the following equation holds: $$a^{p-1} \equiv 1 \bmod p$$ In general this theorem doesn't hold for composite numbers. This can be used to create a primality test. We pick an integer $2 \le a \le p - 2$, and check if the equation holds or not. If it doesn't hold, e.g. $a^{p-1} \not\equiv 1 \bmod p$, we know that p cannot be a prime number. In this case we call the base a a Fermat witness for the compositeness of p . However it is also possible, that the equation holds for a composite number. So if the equation holds, we don't have a proof for primality. We only can say that p is probably prime . If it turns out that the number is actually composite, we call the base a a Fermat liar . By running the test for all possible bases a , we can actually prove that a number is prime. However this is not done in practice, since this is a lot more effort that just doing trial division . Instead the test will be repeated multiple times with random choices for a . If we find no witness for the compositeness, it is very likely that the number is in fact prime. bool probablyPrimeFermat (int n , int iter = 5) { if (n < 4) return n == 2 || n == 3 ; for (int i = 0 ; i < iter ; i ++) { int a = 2 + rand () % (n - 3); if (binpower (a , n - 1 , n) != 1) return false ; } return true ; } We use Binary Exponentiation to efficiently compute the power a^{p-1} . There is one bad news though: there exist some composite numbers where $a^{n-1} \equiv 1 \bmod n$ holds for all a coprime to n , for instance for the number $561 = 3 \cdot 11 \cdot 17$. Such numbers are called Carmichael numbers . The Fermat primality test can identify these numbers only, if we have immense luck and choose a base a with $\gcd(a, n) \ne 1$. The Fermat test is still being used in practice, as it is very fast and Carmichael numbers are very rare. E.g. there only exist 646 such numbers below 10^9 . Miller-Rabin primality test ¶ The Miller-Rabin test extends the ideas from the Fermat test. For an odd number n , $n-1$ is even and we can factor out all powers of 2. We can write: $$n - 1 = 2^s \cdot d,~\text{with}~d~\text{odd}.$$ This allows us to factorize the equation of Fermat's little theorem: $$\begin{array}{rl} a^{n-1} \equiv 1 \bmod n &\Longleftrightarrow a^{2^s d} - 1 \equiv 0 \bmod n \\\\ &\Longleftrightarrow (a^{2^{s-1} d} + 1) (a^{2^{s-1} d} - 1) \equiv 0 \bmod n \\\\ &\Longleftrightarrow (a^{2^{s-1} d} + 1) (a^{2^{s-2} d} + 1) (a^{2^{s-2} d} - 1) \equiv 0 \bmod n \\\\ &\quad\vdots \\\\ &\Longleftrightarrow (a^{2^{s-1} d} + 1) (a^{2^{s-2} d} + 1) \cdots (a^{d} + 1) (a^{d} - 1) \equiv 0 \bmod n \\\\ \end{array}$$ If n is prime, then n has to divide one of these factors. And in the Miller-Rabin primality test we check exactly that statement, which is a more stricter version of the statement of the Fermat test. For a base $2 \le a \le n-2$ we check if either $$a^d \equiv 1 \bmod n$$ holds or $$a^{2^r d} \equiv -1 \bmod n$$ holds for some $0 \le r \le s - 1$. If we found a base a which doesn't satisfy any of the above equalities, then we found a witness for the compositeness of n . In this case we have proven that n is not a prime number. Similar to the Fermat test, it is also possible that the set of equations is satisfied for a composite number. In that case the base a is called a strong liar . If a base a satisfies the equations (one of them), n is only strong probable prime . However, there are no numbers like the Carmichael numbers, where all non-trivial bases lie. In fact it is possible to show, that at most $\frac{1}{4}$ of the bases can be strong liars. If n is composite, we have a probability of $\ge 75\%$ that a random base will tell us that it is composite. By doing multiple iterations, choosing

different random bases, we can tell with very high probability if the number is truly prime or if it is composite. Here is an implementation for 64 bit integer. using u64 = uint64_t ; using u128 = __uint128_t ; u64 binpower (u64 base , u64 e , u64 mod) { u64 result = 1 ; base %= mod ; while (e) { if (e & 1) result = (u128) result * base % mod ; base = (u128) base * base % mod ; e >>= 1 ; } return result ; } bool check_composite (u64 n , u64 a , u64 d , int s) { u64 x = binpower (a , d , n); if (x == 1 || x == n - 1) return false ; for (int r = 1 ; r < s ; r ++) { x = (u128) x * x % n ; if (x == n - 1) return false ; } return true ; }; bool MillerRabin (u64 n , int iter = 5) { // returns true if n is probably prime, else returns false. if (n < 4) return n == 2 || n == 3 ; int s = 0 ; u64 d = n - 1 ; while ((d & 1) == 0) { d >>= 1 ; s ++ ; } for (int i = 0 ; i < iter ; i ++) { int a = 2 + rand () % (n - 3); if (check_composite (n , a , d , s)) return false ; } return true ; } Before the Miller-Rabin test you can test additionally if one of the first few prime numbers is a divisor. This can speed up the test by a lot, since most composite numbers have very small prime divisors. E.g. 88% of all numbers have a prime factor smaller than 100 . Deterministic version ¶ Miller showed that it is possible to make the algorithm deterministic by only checking all bases $\le O((\ln n)^2)$. Bach later gave a concrete bound, it is only necessary to test all bases $a \le 2 \ln(n)^2$. This is still a pretty large number of bases. So people have invested quite a lot of computation power into finding lower bounds. It turns out, for testing a 32 bit integer it is only necessary to check the first 4 prime bases: 2, 3, 5 and 7. The smallest composite number that fails this test is $3,215,031,751 = 151 \cdot 751 \cdot 28351$. And for testing 64 bit integer it is enough to check the first 12 prime bases: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, and 37. This results in the following deterministic implementation: bool MillerRabin (u64 n) { // returns true if n is prime, else returns false. if (n < 2) return false ; int r = 0 ; u64 d = n - 1 ; while ((d & 1) == 0) { d >>= 1 ; r ++ ; } for (int a : { 2 , 3 , 5 , 7 , 11 , 13 , 17 , 19 , 23 , 29 , 31 , 37 }) { if (n == a) return true ; if (check_composite (n , a , d , r)) return false ; } return true ; } It's also possible to do the check with only 7 bases: 2, 325, 9375, 28178, 450775, 9780504 and 1795265022. However, since these numbers (except 2) are not prime, you need to check additionally if the number you are checking is equal to any prime divisor of those bases: 2, 3, 5, 13, 19, 73, 193, 407521, 299210837. Practice Problems ¶ SPOJ - Prime or Not Project euler - Investigating a Prime Pattern Contributors: jakobkogler (91.41%) gampu (3.17%) adamant-pwn (2.71%) boxlesscat (1.36%) pokorj54 (0.45%) siddharthabhi30 (0.45%) HKalbasi (0.45%)