Solovay-Strassen and Miller-Rabin Primality Tests

Kobe Luong

April 16, 2020

Solovay-Strassen Primality Test

Idea: Use theorem(Euler) that if N is prime, then $(\frac{a}{n}) \equiv a^{(n-1)/2} \pmod{n}$, with n being odd.

Steps for Solovay-Strassen(repeat several times):

1. Pick "a," where "a" is a random number greater than 1 but less than n(1<a<n)
2. Check if $(\frac{a}{n}) \equiv a^{(n-1)/2} \pmod{n}$
3. If true, return "probably prime." Else, return "composite."

Example: Use Solovay-Strassen to test n=91

1. Pick a=10
2. Compute $(\frac{10}{91})$
3. $(\frac{10}{91})$ can be broken down into: $(\frac{2}{91})(\frac{5}{91})$
4. $(\frac{2}{91})$, $91 \equiv 3 \pmod{8}$, meaning that $(\frac{2}{91})$ evaluates to -1
5. $(\frac{5}{91})$ can be flipped to $(\frac{91}{5})$
6. You can reduce 91 by (mod 5), turning it into $(\frac{1}{5})$
7. This is overall (-1)(1), which evaluates to -1
8. Compute $10^{(91-1)/2} \equiv 10^{45} \pmod{91} \equiv 10^{32+8+4+1}$
9. $10^2 \equiv 100 \equiv 9 \pmod{91}$
10. $10^4 \equiv 9^2 \equiv 81 \equiv -10 \pmod{91}$
11. $10^8 \equiv (-10)^2 \equiv 100 \equiv 9 \pmod{91}$
12. $10^{16} \equiv 9^2 \equiv -10 \pmod{91}$

13. $10^{32} \equiv (-10)^2 \equiv 9 \pmod{91}$

14. $10^{45} \equiv (10^{32})(10^8)(10^4)(10^1)$

15. $(10^{32})(10^8)(10^4)(10^1) \equiv (9)(9)(-10)(10)$, taken from the corresponding final answers from the modular exponentiation steps we just did.

16. The (9)(9) from step 15 is the same as 9^2, which corresponds to the final answer from step 12, which is -10. Overall, you now have (-10)(-10)(10)

17. The (-10)(-10) from step 16 is just 100, which corresponds to the final answer of 9 from step 11. You now just have $(9)(10) \equiv 90 \pmod{91}$

18. Check if $-1 \equiv \frac{10}{91} \equiv 10^{45} \pmod{91}$

19. It turns out that the equation from step 18 is congruent to 90 (mod 91).

20. This means that the Solovay-Strassen test will say that 91 is probably prime.

21. Now let's try a=11

22. Compute $(\frac{11}{91}) = -(\frac{91}{11}) = -(\frac{3}{11}) = +(+(\frac{11}{3})) = (\frac{11}{3}) = (\frac{2}{3}) = -1$

23. Compute $11^{(91-1)/2} \equiv 11^{45} \pmod{91} \equiv 11^{32+8+4+1}$

24. $11^2 \equiv 121 \equiv 30 \pmod{91}$

25. $11^4 \equiv 30^2 \equiv 900 \equiv -10 \pmod{91}$

26. $11^8 \equiv (-10)^2 \equiv 9 \pmod{91}$

27. $11^{16} \equiv 9^2 \equiv -10 \pmod{91}$

28. $11^{32} \equiv (-10)^2 \equiv 9 \pmod{91}$

29. $(11^{32})(11^8)(11^4)(11^1) \equiv (9)(9)(-10)(11)$, taken from the corresponding final answers from the modular exponentiation steps we just did.

30. The (9)(9) from step 29 is the same as 9^2, which corresponds to the final answer from step 27, which is -10. We now have (-10)(-10)(11)

31. The (-10)(-10) correspond to the final answer from step 26, which is 9. We now have (9)(11)

32. $(9)(11) \equiv 99 \equiv 8 \pmod{91}$

33. Check if $-1 \equiv \frac{11}{91} \equiv 8 \pmod{91}$

34. False! So return "composite."

Programming this Algorithm Out

The calculation of the Jacobi Symbols and the repeated squaring take up notable computational power, making the time complexity of this algorithm's worst-case scenario $O(k * (ln(n))^2)$, where k is the amount of times you repeat the steps and n is the number being checked for primality.

Here is an example of C++ Solovay-Strassen code I got from geeksfromgeeks.org. I only modified it a little bit:
https://www.geeksforgeeks.org/primality-test-set-4-solovay-strassen/

C++ Code

```
// C++ program to implement Solovay-Strassen
// Primality Test
#include <iostream>
using namespace std;

// modulo function to perform binary exponentiation
long long modulo(long long base, long long exponent,
long long mod)
{
long long x = 1;
long long y = base;
while (exponent > 0)
{
if (exponent % 2 == 1)
{
x = (x * y) % mod;
}
y = (y * y) % mod;
exponent = exponent / 2;
}

return x % mod;
}

// To calculate Jacobian symbol of a given number
int calculateJacobian(long long a, long long n)
{
if (!a)
{
return 0;// (0/n) = 0
}
int ans = 1;
if (a < 0)
{
```

```
a = -a; // (a/n) = (-a/n)*(-1/n)
if (n % 4 == 3)
{
ans = -ans; // (-1/n) = -1 if n = 3 (mod 4)
}
}

if (a == 1)
{
return ans;// (1/n) = 1
}
while (a)
{
if (a < 0)
{
a = -a;// (a/n) = (-a/n)*(-1/n)
if (n % 4 == 3)
{
ans = -ans;// (-1/n) = -1 if n = 3 (mod 4)
}
}

while (a % 2 == 0)
{
a = a / 2;
if (n % 8 == 3 || n % 8 == 5)
{
ans = -ans;
}
}

swap(a, n);

if (a % 4 == 3 && n % 4 == 3)
{
ans = -ans;
}
a = a % n;

if (a > n / 2)
{
a = a - n;
}

}
```

```
if (n == 1)
{
return ans;
}

return 0;
}

// To perform the Solovay-Strassen Primality Test
bool solovoyStrassen(long long p, int iterations)
{
if (p < 2)
{
return false;
}
if (p != 2 && p % 2 == 0)
{
return false;
}

for (int i = 0; i < iterations; i++)
{
// Generate a random number a
long long a = rand() % (p - 1) + 1;
long long jacobian = (p + calculateJacobian(a, p)) % p;
long long mod = modulo(a, (p - 1) / 2, p);

if (!jacobian || mod != jacobian)
{
return false;
}
}
return true;
}

// Driver Code
int main()
{
int iterations = 50;
long long num1 = 91;
long long num2 = 101;

if (solovoyStrassen(num1, iterations))
{
printf("%d is prime\n", num1);
}
```

```
else
{
printf("%d is composite\n", num1);
}

if (solovoyStrassen(num2, iterations))
{
printf("%d is prime\n", num2);
}
else
{
printf("%d is composite\n", num2);
}

return 0;
}
```

Miller-Rabin Primality Test

- "Souped up" Fermat Primality Test
- Still compute a^{n-1} (mod n)
- Know that if n is prime the final answer has to be 1. Use extra steps.

Theorem(Factoring Trick): If n > 1 and

- $a^2 \equiv b^2$ (mod n),
- but a !≡ b (mod n)
- and a !≡ -b (mod n)
- (find 2 numbers that square to the same number but not for an obvious reason),

then n is composite and gcd(n, b-a) is a nontrivial factor of n.

Note: A nontrivial factor is a number that divides n besides 1 and n.

Proof(by contradiction)

Suppose n>1

- and $a^2 \equiv b^2$ (mod n)
- but a !≡ b (mod n)

- and $a \equiv -b$ (mod n)

Suppose for contradiction that n is a prime number.

Since $a^2 \equiv b^2$ (mod n)

- $a^2 - b^2 \equiv 0$ (mod n)
- (a+b) * (a-b)(n divides this) $\equiv$ 0(n divides this) (mod n)

This only happens if n divides (a+b) or n divides (a-b)(since n is prime). If n divides (a+b) that means a+b $\equiv$ 0 (mod n) and a $\equiv$ -b(mod n) This is not allowed to happen.

So n divides (a-b)

- then (a-b) $\equiv$ 0 (mod n)
- a $\equiv$ b (mod n)

This isn't allowed to happen either. This is a contradiction, so n can't be prime! One factor of n divides (a+b) and another factor divides (a-b)

Miller-Rabin Algorithm

One iteration(Choose a random number "a" that is 1<a<n)

Compute $(a^{n-1}) \equiv (((a^m)^2)^2)^2$ (mod n), with k being the 3 exponents that are 2s. Write n-1=m2^k, with m being odd.

1. Compute $b_0 \equiv a^m$ (mod n). If $b_0 \equiv \pm 1$ (mod n), return "probably prime"
2. For i from 1 to k
   compute $b_i \equiv (b_{i-1})^2$ (mod n)
   (note $b_k \equiv a^{n-1}$ (mod n))
   If $b_i \equiv -1$ (mod n), return "probably prime"
   If $b_i \equiv 1$ (mod n), return "composite"
3. If we finish for-loop and the final answer isn't 1, return "composite"
   Why return composite if $b_i \equiv 1$ (mod n)?
   If we got here we haevn't yet seen $\pm$ 1 so b_{i-1} !$\equiv \pm 1$ (mod n)
   But 1(also equal to 1^2) $\equiv b_i \equiv (b_{i-1})^2$ (mod n)
   So $1^2 \equiv (b_{i-1})^2$ (mod n), but b_{i-1} ! $\equiv \pm 1$ (mod n)
   By factoring trick n has to be composite!

- Like Solovay Strassen the Miller-Rabin test has no "Carmichael numbers"
- Any composite number has witnesses with Miller-Rabin.

- In fact at least 3/4 of the choices for a tell the truth
- If we do this k times the probability we're wrong is at most $4^{-k} = 2^{-2k}$<- Converges to 0 twices as fast Solovay Strassen.

Example: n=35, test using Miller-Rabin

Pick a=b write n-1 = 34 = 2^1*17, k=1 and m=17

1. Compute 6^m(mod 35)
   $6^{17} \equiv 6^{16} * 6^1$
   $6^2 \equiv 36 \equiv 1$ (mod 35)
   $6^4 \equiv 1$ (mod 35)
   ...
   $6^{16} \equiv 1$(mod 35)
   So $b_0 \equiv 6^{17} \equiv (6^{16})(6^1) \equiv 1(6) \equiv 6$ (mod 35)
   Not ± 1 (mod 35)
   Proceed to step 2!

2. Compute: $b_1 = (b^0)^2 \equiv 6^2 \equiv 36 \equiv 1$ (mod 35)
   return "composite"
   Note that Fermat Test would have returned "probably prime" since $6^{34} \equiv 1$ (mod 35)

- The worst-case time complexity for this algorithm is $O(k * ln(n)^2)$, where k is the number of tests and n is the number being tested for primality.
- In practice Miller-Rabin is used for RSA. Miller-Rabin could still be wrong.

Here is an example of C++ Miller-Rabin code I got from geeksfromgeeks.org. I only modified it a little bit:
https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/

C++ Code

```
// C++ program Miller-Rabin primality test
#include <iostream>
using namespace std;

// Utility function to do modular exponentiation.
// It returns (x^y) % p
int power(int x, unsigned int y, int p)
{
int res = 1;      // Initialize result
```

```
x = x % p;  // Update x if it is more than or
// equal to p
while (y > 0)
{
// If y is odd, multiply x with result
if (y & 1)
{
res = (res * x) % p;
}
// y must be even now
y = y >> 1; // y = y/2
x = (x * x) % p;
}
return res;
}

// This function is called for all k trials. It returns
// false if n is composite and returns false if n is
// probably prime.
// d is an odd number such that  d*2<sup>r</sup> = n-1
// for some r >= 1
bool miillerTest(int d, int n)
{
// Pick a random number in [2..n-2]
// Corner cases make sure that n > 4
int a = 2 + rand() % (n - 4);

// Compute a^d % n
int x = power(a, d, n);

if (x == 1 || x == n - 1)
{
return true;
}

// Keep squaring x while one of the following doesn't
// happen
// (i)   d does not reach n-1
// (ii)  (x^2) % n is not 1
// (iii) (x^2) % n is not n-1
while (d != n - 1)
{
x = (x * x) % n;
d *= 2;

if (x == 1)
```

```
{
return false;
}
if (x == n - 1)
{
return true;
}
}

// Return composite
return false;
}

// It returns false if n is composite and returns true if n
// is probably prime.  k is an input parameter that determines
// accuracy level. Higher value of k indicates more accuracy.
bool isPrime(int n, int k)
{
// Corner cases
if (n <= 1 || n == 4)
{
return false;
}
if (n <= 3)
{
return true;
}

// Find r such that n = 2^d * r + 1 for some r >= 1
int d = n - 1;
while (d % 2 == 0)
{
d /= 2;
}

// Iterate given nber of 'k' times
for (int i = 0; i < k; i++)
{
if (!miillerTest(d, n))
{
return false;
}
}
return true;
}
```

```
// Driver program
int main()
{
int k = 100;  // Number of iterations
int n = 35;
cout << "All primes smaller than 35: \n";

if (isPrime(n, k))
{
cout << n << " is probably prime.";
}
else
{
cout << n << " is composite.";
}

return 0;
}
```

- Is there a first test(polynomial time) that tells us for sure if a number is prime?
  The AKS test will tell us for certain but its time complexity for the worst-case scenario is O($(log(n))^{11}$). It's way too slow to be practical.

- There is a conjecture called the Riemann hypothesis. If it is true, then lots of cool things can be proven. One of those is that the Miller-Rabin test always has a witness less than $2ln(n)^2$.

- If you believe the Riemann hypothesis, then you can prove a number is prime by testing all a's up to $2ln(n)^2$, and then it would only be called Miller's Test!

- The worst-case time complexity of this algorithm would be O($ln(n)^4$)